# Software Productivity Consortium

# Overview of Megaprogramming Course: Lectures and Exercises

AD-A286 838

DTIC
SELECTED
AUG 21 1995

B

95006-CMC
Version 01.00.07

95-01831

February 1995

# Overview of Megaprogramming Course: Lectures and Exercises
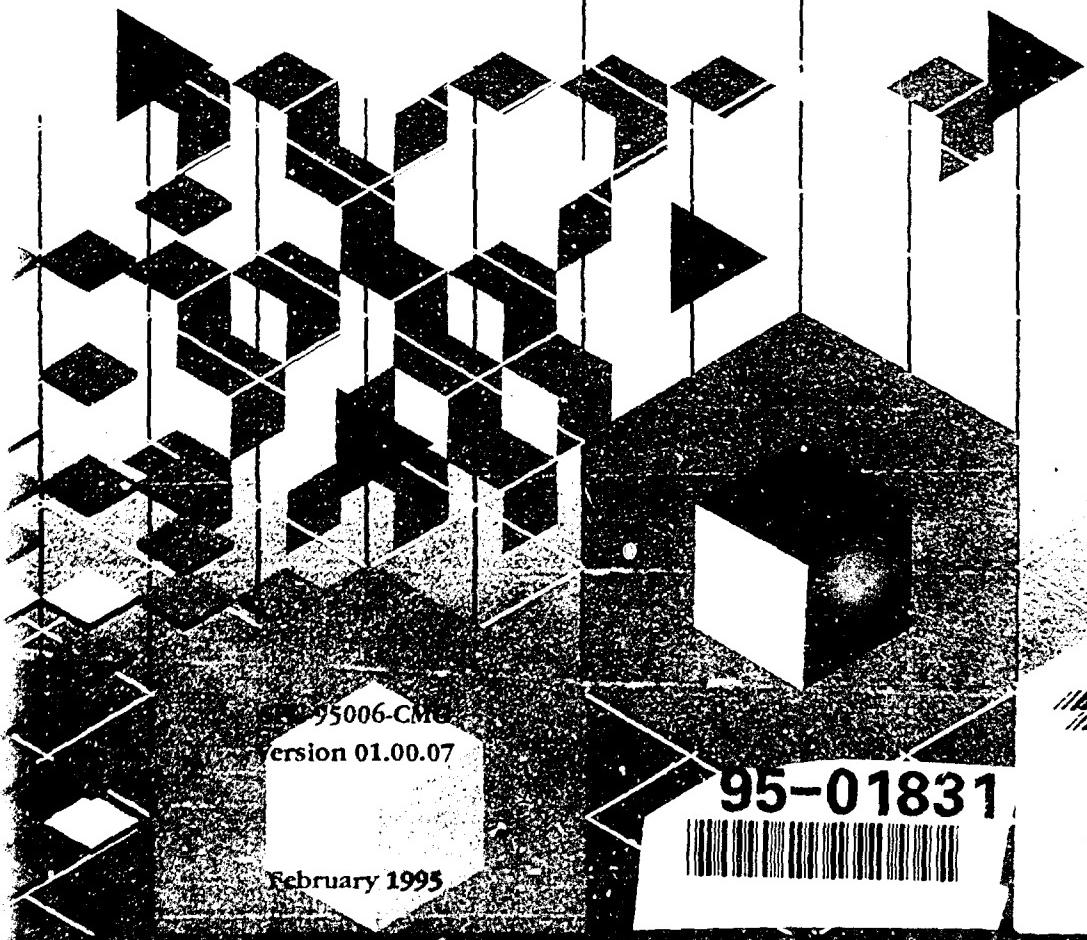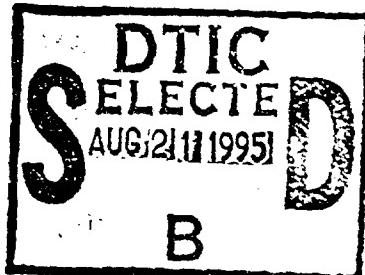
## SPC-95006-CMC

## Version 01.00.07

## February 1995

95 C18 005

# REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | February 1995 | Technical Report - Final |

| 4. TITLE AND SUBTITLE | 5. FUNDING NUMBERS |
|---|---|
| Overview of Megaprogramming Course: for Macintosh | |
| 6. AUTHOR(S) R. Christopher, M. Eward, S. Wartik<br>Produced by Software Productivity Consortium under contract<br>to Virginia Center of Excellence | G MDA972-92-J-1018 |

| 7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES) | 8. PERFORMING ORGANIZATION REPORT NUMBER |
|---|---|
| Virginia Center of Excellence<br>SPC Building<br>2214 Rock Hill Road<br>Herndon, VA 22070 | SPC-95003-CMC,<br>Version 01.00.00 |

| 9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSORING / MONITORING AGENCY REPORT NUMBER |
|---|---|
| ARPA/SISTO<br>Suite 400<br>801 N. Randolph Street<br>Arlington, VA 22203 | |

**11. SUPPLEMENTARY NOTES**

This course includes the following four units. Overview of Megaprogramming Course: Lectures and Exercises (SPC-95006-CMC); Teacher Notes for Overview of Megaprogramming Course (SPC-94044-CMC); Diskette-Overview of Megaprogramming Course: Application Engineering Environment for Macintosh (SPC-95005-CMC); Installation Guide (SPC-95004-CMC)

| 12a. DISTRIBUTION / AVAILABILITY STATEMENT | 12b. DISTRIBUTION CODE |
|---|---|
| No Restrictions | 1 |

**13. ABSTRACT** (Maximum 200 words)

This is a short course that introduces megaprogramming concepts. The course, which takes between 1 to 2 weeks to teach, is aimed at high school students who have a basic understanding of programming. The course discusses "programming-in-the-large" concepts not usually covered in introductory classes, but which are increasingly being seen as fundamental. The material focuses on software reuse and software process. The stress on the former shows students the value of creating reusable software. The stress on the latter establishes a context for reuse and other topics that will be encountered in future, more advanced courses.

The course material contains viewgraphs which can serve as the basis of lectures. Each viewgraph has accompanying instructor notes (SPC-94044-CMC), which identify the important points on the viewgraph and suggest topics for discussion. The course is divided into 4 units; following each unit are summaries and homework assignments. In addition, there are a laboratory exercise for Unit 3 (supported by software for Macintosh compatible computers (SPC-95005-CMC), an installation guide for the laboratory (SPC-95004-CMC), a test following Unit 4, and an evaluation form.

| 14. SUBJECT TERMS | | 15. NUMBER OF PAGES |
|---|---|---|
| Megaprogramming, software reuse, software process, course | | 110 |
| | | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |

# CONTENTS

*This page intentionally left blank.*

*Virginia*

# *CENTER of*
# *EXCELLENCE*

*for Software Reuse and Technology Transfer*

# Overview of Megaprogramming Course

# February 1995

To set the stage for the course, ask your students to write down their answers to the following questions:

- How do you write a software program now?

- How do you think you **should** write a software program?

This course will teach your students a better way to perform software development.

*******************************************************************

## DISCUSSION

You are learning key computer science principles as you learn how to write software programs in Pascal.

Software development involves more than just writing the code.

The traditional approach to software development has a lot of limitations and problems associated with it.

There's a technique that is different from the traditional software development process. It's called megaprogramming.

- For the teacher: Megaprogramming is in the research stage, although individual ideas within megaprogramming have been in use within industry for years. Because megaprogramming is in its infancy, some of the terms and some of the specific details may change. The underlying concepts and principles behind megaprogramming are unlikely to change.

Today, we want to explain traditional software development concepts and introduce megaprogramming.

First, we want you to understand why you should care about megaprogramming.

## OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Explain the motivation for megaprogramming

- State that megaprogramming looks at similar problems and solutions together, as opposed to seeing each problem/solution combination as unique

*virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology T*

# Unit 1: Software Development

## DISCUSSION

Computer programming started with machine language, then assembly language, then simple third generation languages (3 GLs—BASIC, FORTRAN, and COBOL), then more complex 3 GLs (Pascal, C, and Ada), then fourth generation languages (spreadsheets, HyperCard, Visual Basic). Megaprogramming is another big step: this may be how most software is developed in the future.

Megaprogramming sees generating software as a process of defining and solving problems, rather than just a programming process. This concept is more in line with what you will see if you get a job involving problem solving with computers.

Because megaprogramming deals with more than just programming, you first need to understand the "big picture" of how software is typically developed.

## STUDENT INTERACTIONS

- How many of you have ever programmed in a language other than what you are using in this class? Was it better or worse? Why? These answers should support how each subsequent generation has been better than previous generations.

- Does anybody know what machine language is? Assembly language? Have you written any programs using either? How does it compare to the programming language you have used in this course?

- In movies and TV shows that take place in the future (e.g., Star Trek), what did the computers do? Do you think we could program computers to work like that by programming the way we do now?

- Have you ever thought that developing software would be easier if you possessed some missing knowledge or capability? What was it? Have you ever wished for some software to help you develop software? What type of software would it be?

## OBJECTIVE

The students should be able to:

- Understand that megaprogramming is something about which they should be concerned

# Evolution of Software Development

1950s

1990s ...

**Machine Language**

↳ **Assembly Language**

↳ **Early 3GLs (FORTRAN, BASIC)**

↳ **Advanced 3GLs (Ada, C, Pascal)**

↳ **4GLs (Spreadsheets)**

↳ **MEGAPROGRAMMING**
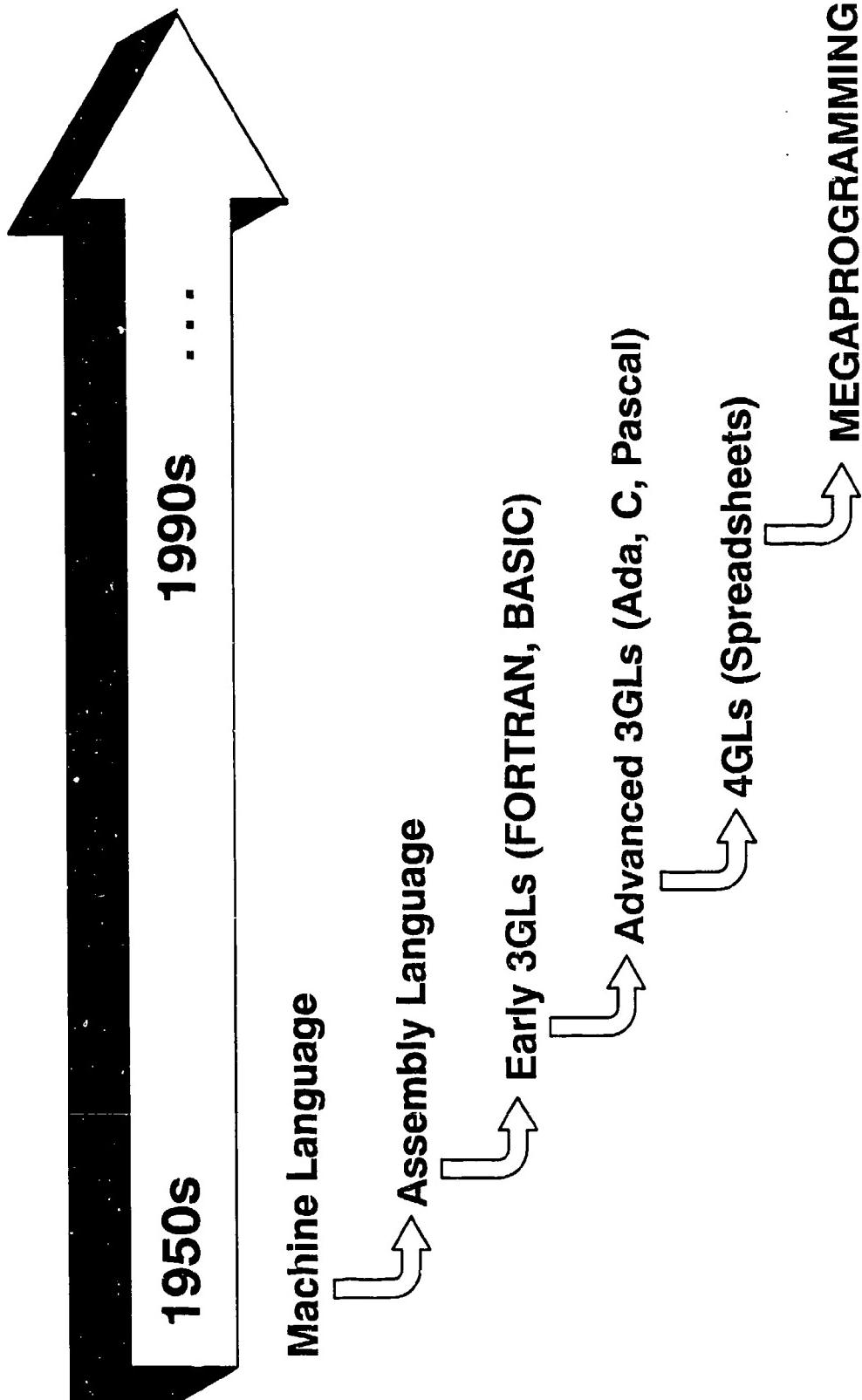
## DISCUSSION

Software is built and used following what industry terms a *life cycle*. A software life cycle describes the key steps in developing and supporting software.

These steps usually include the steps shown on this picture.

- Statement of Need. Your customer asks you to develop a software system that solves some problem.

- Software Development Process. You develop the software system that solves the problem.

- Operation and Maintenance. Your customer uses the software system. You fix any bugs that are found; you add, modify, and delete features as the customer's needs change.

- Retirement. Your customer stops using the software because (1) the software no longer supports a need or (2) a new version of the software is developed that replaces what the customer has been using or makes it obsolete. The software is therefore not used any more.

We are focusing on the second step, the Software Development Process, in this course.

How has software traditionally been developed?

## STUDENT INTERACTIONS

- Do you think that the life cycle for software is the same for other nonsoftware products (e.g., automobiles)? Why?

- Can you name a software product that has been retired? Why was it retired? Examples of retired software products are old MS-DOS versions and software that ran on obsolete computers.

## OBJECTIVES

The students should be able to:

- State that the software developer is always meeting a customer's needs by writing software (currently, the teacher is the customer)

- State that the customer must be supported during use of the software (that is, the job doesn't end once the software is written)

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Tr...*

# The Big Picture of Software Development

## Software Life Cycle

**Statement of Need**

Customer → Need → Software Developer

**Software Development Process**

**Operation and Maintenance**

Customer → Software and Support → Software Developer

Customer → Needs → Software Developer

**Retirement**
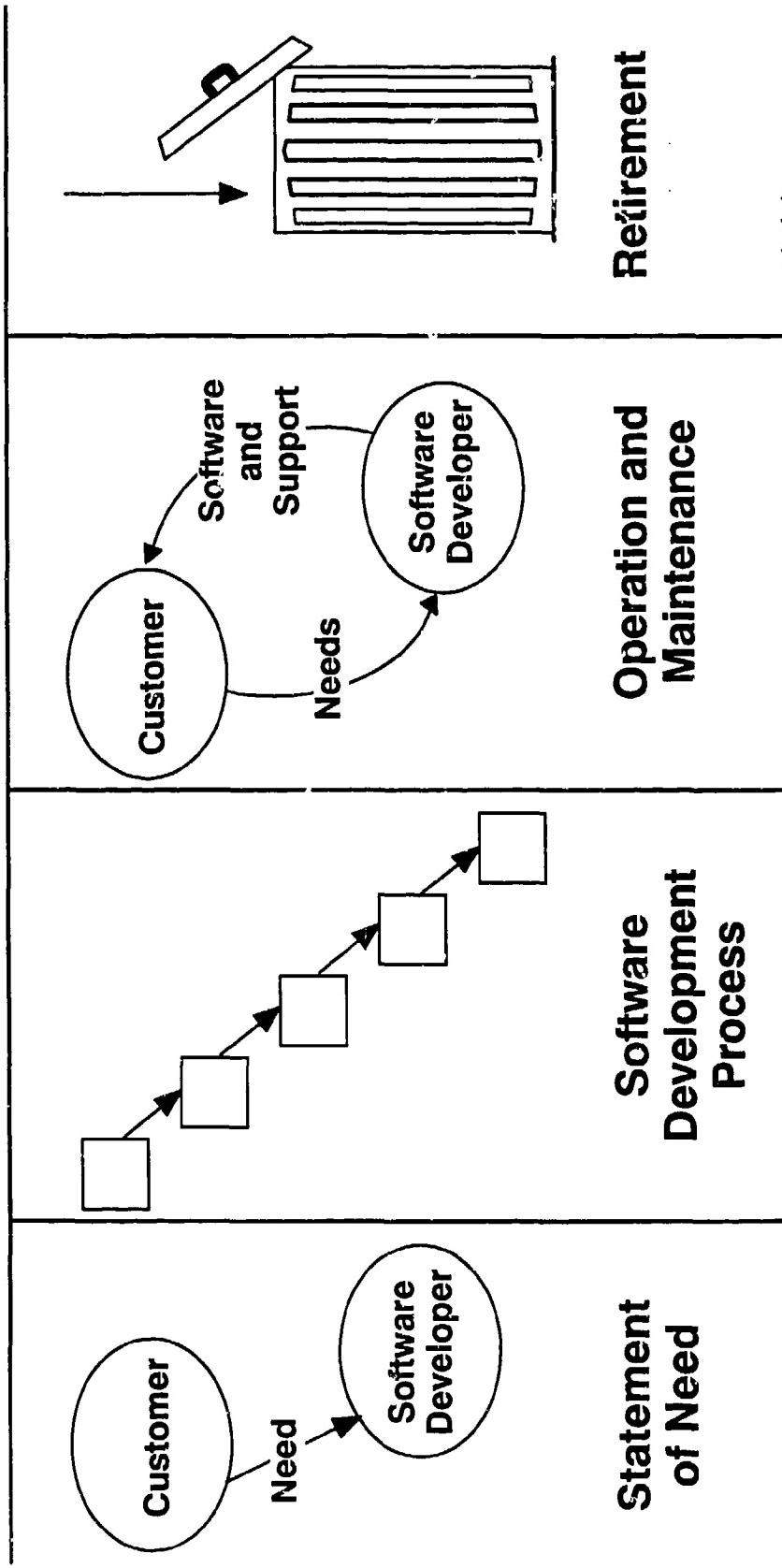
## DISCUSSION

This is the way software is traditionally developed.

This relates to the programming you've been doing in class:

- Your teacher gives you problems to understand and solve: these are the requirements. In industry, engineers take the customer's general statement of need and translate it into more specific requirements.

- You figure out what procedures you need and the calling hierarchy among them: this is the design.

- You write the code and the documentation.

- You make sure your program solves the problem your teacher gave you: this is testing.

- You turn your homework in: this is delivering the software.

In industry, software is usually developed by teams of engineers. That is, different parts of the entire software system are developed by different teams. During the design phase, the engineers managing the development need to plan carefully how the system is parceled into these different parts (these parts include software and documentation). The engineers then need to make sure that: (1) all of the parts developed by the different teams fit together into one working system; (2) the entire system is tested; and (3) the system is delivered to the customer and supported during use.

Now that you understand some of the "big picture," we can give you some more motivation as to why mega-programming is needed in the computer industry.

## STUDENT INTERACTIONS

- Must the steps be done in this order? Can you omit one and still develop good software?

## OBJECTIVE

The students should be able to:

- Define the key steps in a traditional software development process

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Tr*

# The Traditional
# Software Development Process

**Requirements**

Define the problem.

**Design**

Plan how the software will be organized.

**Code**

Implement the plan by writing the code.

**Test**

Test the software to make sure it works.

**Deliver**

Deliver the software to the customer.

**Support**

Support customer use of the software.

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Transfer*
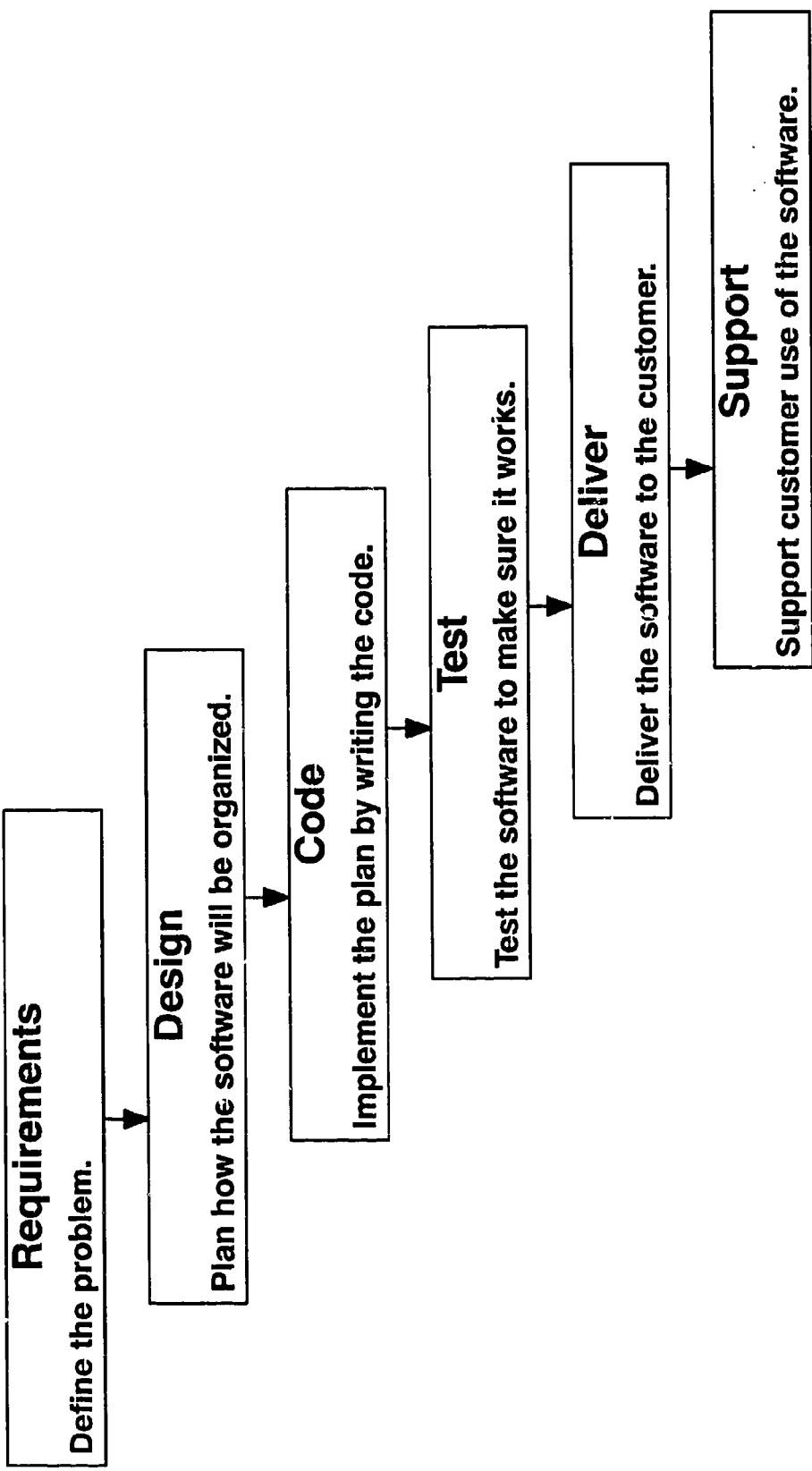
## DISCUSSION

The chart on the left shows the impact if you don't catch a mistake until later in the software development process: it's four times more expensive to fix it in testing and 100 times more expensive to fix it in maintenance than it is to fix it in the requirements step. This is for the SAME mistake.

Obviously, emphasis on doing things right belongs in the early steps of the software development process (i.e., the requirements and design steps).

In fact, projects usually spend more time in requirements and design than they do in coding. The chart on the right shows percentage of time spent on each of the steps for a typical project in industry. Notice that on a typical project you will only spend 20% of the time on coding. Notice that twice as much time is spent both in requirements and design and in testing than on doing the actual coding.

Together, these charts show that you should, and do, spend a lot of time defining the requirements. In fact, software requirements are the most important step in the process. As we'll see, megaprogramming emphasizes getting the requirements right at the start. This is a valuable contribution. In the traditional software development process, customers often don't discover problems in their requirements until they are using the software. In other words, they have the right solution to the wrong problem. This is no better than having the wrong solution to the right problem!

Why are software requirements so important?

## STUDENT INTERACTIONS

- How much time do you spend in design versus testing versus coding?

- Why do you think it costs so much more to fix a bug in maintenance than it does in requirements?

## OBJECTIVES

The students should be able to:

- State that writing the code takes up less time than the other steps

- Explain why it is necessary to spend more time in the requirements step

# Some Data on Software Development

Testing
40%

Coding
20%

Requirements
and Design
40%
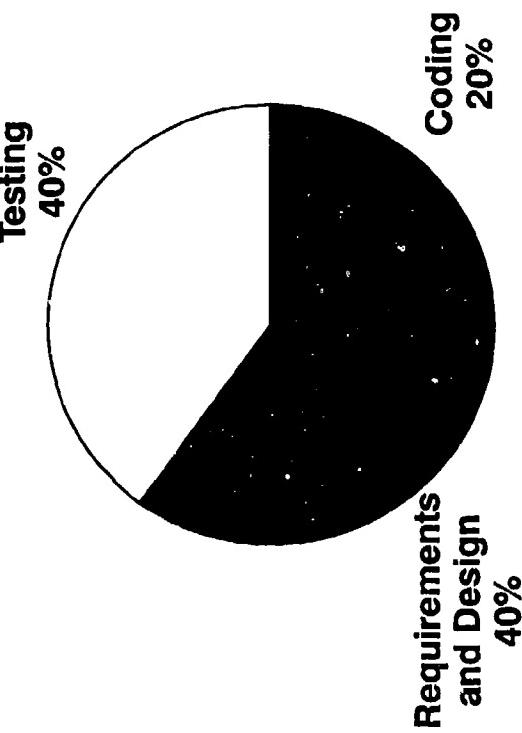
**Percentage of Time Spent
in Different Software
Development Steps**

Requirements    | 1x

Testing    | 4x

Maintenance    100x

0  20  40  60  80  100
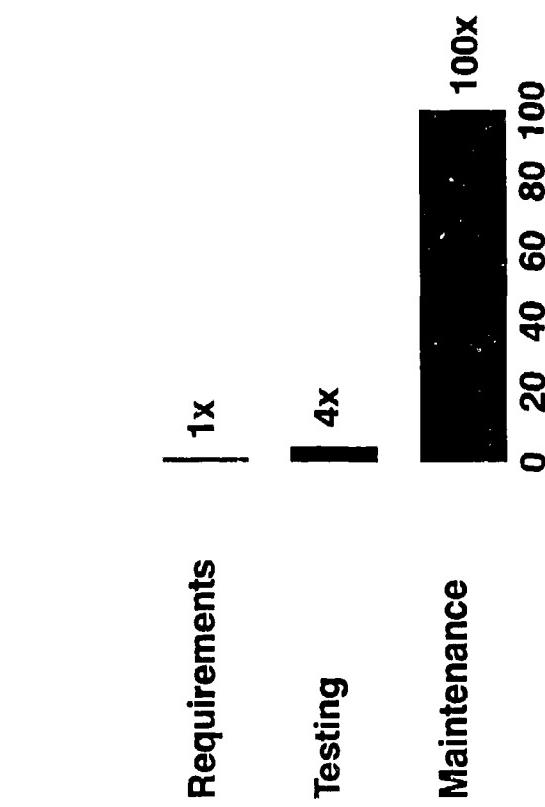
**Relative Cost to Fix the SAME
Mistake in Different Software
Development Steps**

## DISCUSSION

Requirements define the problem: they tell you what the software needs to do. The rest of the steps in the traditional software development process create the solution. To create requirements, an engineer takes the customer's general statement of need and turns it into a specific description of what the software should do.

If you don't know exactly what the software is supposed to do (that is, if the requirements are wrong or vague), then how can you do the right thing?

Take the examples on this slide. The bad (vague and untestable) requirements can be interpreted differently by different people (and usually are). What, exactly, does "responsive" mean? What the developer thinks is "responsive" is probably not what the customer thinks is responsive.

Wrong or vague requirements are responsible for some of the biggest problems industry has with the traditional process (in fact, problems in requirements are often cited as industry's #1 software problem):

- Customers often don't know what they really need.
- Customers' actual needs change (even as the software is being developed).
- Customers do not communicate their needs to software developers clearly and completely.

Requirements are hard to write down: they need to be complete, cover all of the details, and work together. This becomes nearly impossible when the number of detailed requirements gets up into the **thousands.** The exercise after this unit will help you understand how hard it is to write detailed, specific requirements.

Megaprogramming helps you in the requirements stage. How does megaprogramming do this?

## STUDENT INTERACTIONS

- What have been your requirements for the programs you've been writing in this class? How important have they been to you in writing the right program?

- Which of the requirements in the right column are still vague? How can they be improved?

## OBJECTIVE

The students should be able to:

- Explain why the requirements step is the most important step of the software development process

# Software Requirements

## Examples of Bad (Vague and Untestable) Requirements

The system shall be responsive.

The system shall be user-friendly.

The system shall be reliable.

*The best design and the best coding will not help if the requirements are wrong.*

## Examples of Good (at Least Better) Requirements

The system shall provide the correct response to 99.9% of user inputs.

The system shall provide a response within 1 second to 95% of user inputs.

The system user interface shall employ a graphical user interface and permit input via a mouse.

The system shall run uninterrupted for at least 1,000 hours during normal use.

*Are any of these requirements still vague?*

**Virginia**
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Transfer*

1-7

## DISCUSSION

This is a top-level view of the megaprogramming process.

Megaprogramming doesn't look at one problem/solution combination in isolation. Megaprogramming looks at a whole set of problem/solution combinations that are similar in some way. For now, we will call this set a **problem area**.

The purpose of the step above the dotted line is to: (1) define the problem area; and (2) create software and documentation that can be used to create a solution for any particular problem in that problem area.

The purpose of the step below the dotted line is to create a solution for a customer who has a problem in that area. To do this, you use and build on the work done by the engineers who worked to define the problem area.

Everything above the dotted line is done once and then continually improved. Everything below the dotted line is done every time you have to solve a problem in that problem area.

A good analogy to megaprogramming is the building of computers. Most computer companies build several types of computers—for example, desktop and laptop. They don't build all the individual chips and boards differently. They take existing parts and, based on their knowledge of how to build computers, assemble them in different ways depending on what they want to produce. Software developers, on the other hand, traditionally generate software from scratch each time. Megaprogramming is an attempt to allow software developers to do what hardware engineers do: use existing, proven components each time they build a product.

More on this tomorrow (and for the rest of this course).

## STUDENT INTERACTIONS

- Does the megaprogramming analogy for computers relate to the building of cars? When the automobile builder builds a car, do they build everything from scratch?

## OBJECTIVE

The students should be able to:

- Explain that megaprogramming deals with problem areas (a set of similar problems and solutions)

# Megaprogramming

Understand all aspects of a problem area (a set of similar problems with their solutions).

→ Software and documentation to help define and solve a problem in that area

→ Create a solution to a problem in the problem area.

→ The Solution

→ Your Customer

This step defines a problem area.

This step defines a specific problem and creates a solution.

# UNIT 1: SOFTWARE DEVELOPMENT

## SUMMARY

Software development involves more than just writing code.

### SOFTWARE LIFE CYCLE

- The customer states the NEED for the software.

- The developer DEVELOPS the software.

- The software is USED, debugged, and enhanced.

- The software becomes obsolete and is RETIRED.

### SOFTWARE DEVELOPMENT PROCESS

| **Requirements** |
| Define the problem. |

↓

| **Design** |
| Plan how the software will be organized. |

↓

| **Code** |
| Implement the plan by writing the code. |

↓

| **Test** |
| Test the software to make sure it works. |

↓

| **Deliver** |
| Deliver the software to the customer. |

↓

| **Support** |
| Support customer use of the software. |
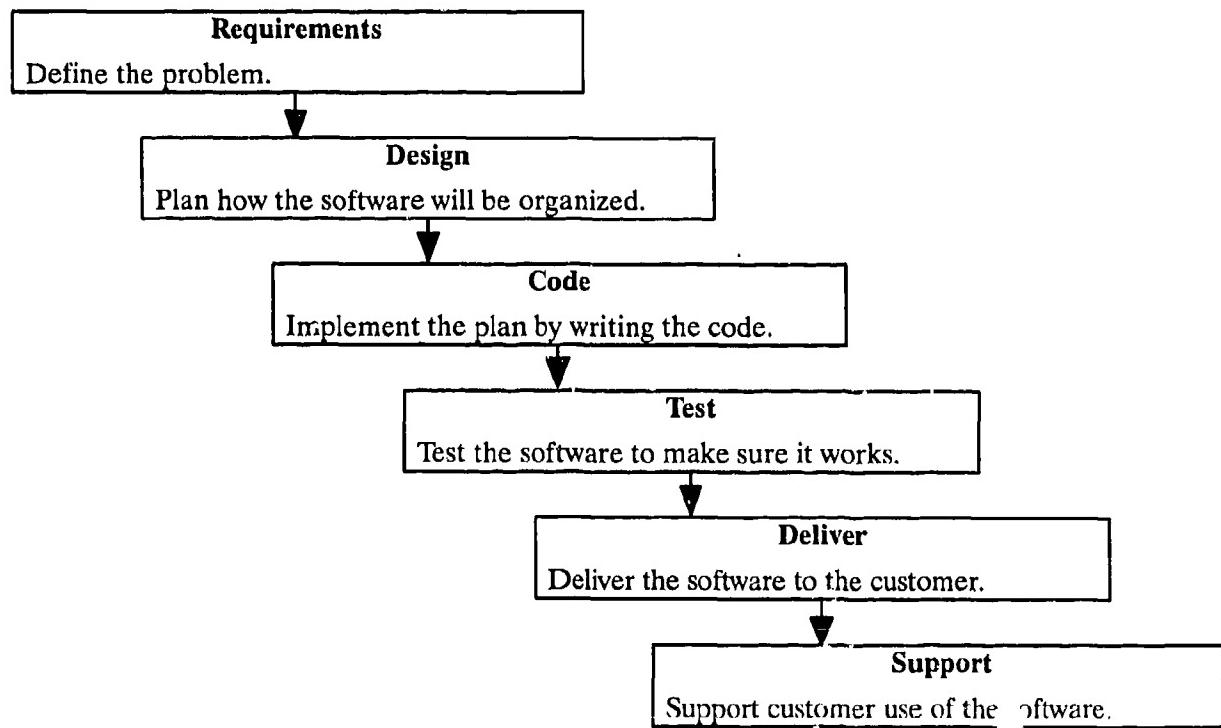
### REQUIREMENTS

- Requirements define the problem.

- Since you cannot solve a problem unless you know what the problem is, defining the requirements is the most important step in software development.

- Wrong or vague requirements cost money.

## MEGAPROGRAMMING

- Megaprogramming is the next generation in software development processes.

- Megaprogramming looks at similar problems and solutions as opposed to seeing each as unique.

- This set of similar problems with their solutions is called a *problem area*.

- Megaprogramming takes advantage of the similarities and differences between the problems when generating a solution to a specific problem.

- In the following figure, everything above the dotted line defines problems and solutions within the problem area. The steps below the dotted line are done to create a specific solution for a specific problem within that problem area.
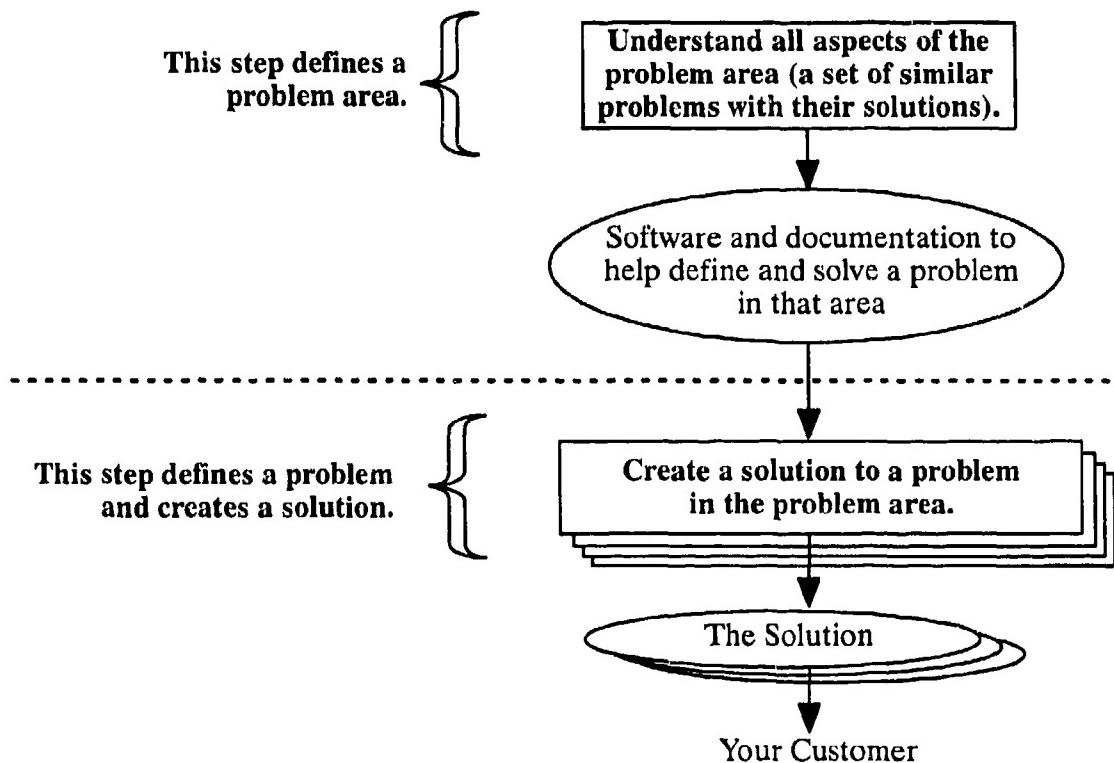
**This step defines a problem area.** Understand all aspects of the problem area (a set of similar problems with their solutions).

Software and documentation to help define and solve a problem in that area

**This step defines a problem and creates a solution.** Create a solution to a problem in the problem area.

The Solution

Your Customer

A good analogy to megaprogramming is the building of computers. Most computer companies build several types of computers—for example, stationary and laptop. They don't build all the individual chips and boards differently. They take existing parts and, based on their knowledge of how to build computers, assemble them in slightly different ways depending on what they want to produce. Software developers, on the other hand, traditionally generate software from scratch each time. Megaprogramming is an attempt to allow software developers to do what hardware engineers do: use existing, proven components each time a product is created.

# UNIT 1: SOFTWARE DEVELOPMENT

## EXERCISES

### GENERATING REQUIREMENTS

Write down all of the information you think you would need to develop a software program that would solve the following problems. Don't worry about specific procedures. List only all of the information you would need to solve the problem.

1.   Beach Trip — You and a friend want to drive to the beach for a weekend and you want to know (1) how long it is going to take and (2) how much the gas is going to cost.

2.   Scheduler — You want to develop a program that automatically schedules all of your activities during the week. You want to be able to run this program every Sunday so you know the time, date, and location for each activity.

3.   The Cleaning Robot — With such busy lives these days, you decide to develop a robot that will clean up litter in a teenager's room.

4.   Several Robots — Suppose you work for United Robot Workers, Inc. (URW). Three customers approach you. Each has different needs:

   a.   Customer 1, a farmer, owns a large cornfield and has trouble finding time to harvest it. She wants to know if you can provide a robot that will harvest her corn without human supervision.

   b.   Customer 2 is from the Alaska National Guard, which is constantly rescuing people who wander too far afield in the tundra. Mounting a rescue party is time consuming; people have died while the members of the party are gathering. The Guard thinks having robots ready could eliminate these life-threatening delays.

   c.   Customer 3, from the National Park Service, is concerned about growing amounts of litter in national parks and wants to know if you can provide a robot that can pick up the litter.

   These three statements correspond to the customers' vague understandings of their problems and of potential solutions. Your task is to write a set of questions for each customer that would clarify each of the problems.

5.   Vending machines — The Student Government Association (SGA) has funds to build a vending machine room near the central hall. The principal has agreed to let the SGA go ahead if they make provisions to keep it attractive and litter free. It is your job to define the requirements for the vending machine. What information do you need to define the requirements?

   List the exact requirements for the particular vending machine you were assigned in class. The requirements you come up with will most likely expand beyond the requirements identified in the class discussion.

*This page intentionally left blank.*

# UNIT 1: SOFTWARE DEVELOPMENT

## TEACHER NOTES FOR EXERCISES

*Here are lists of needed information for each of the problems. Each list is probably not complete. Again, the point of the exercise is not to create a comprehensive list but to make the students realize how hard it is to generate a complete list.*

*Several of these exercises deal with hardware as opposed to software. However, the course lecture material focuses on software. The point of the exercises for all units is to get across key concepts in software development and in megaprogramming. We have used examples in these exercises that we feel will get the concepts across to the students without worrying about whether or not the example was software-related.*

*The first three exercises are optional. The fourth exercise begins the introduction of what the students will see in the laboratory. The fifth exercise is threaded into the next day and can be used as homework.*

### GENERATING REQUIREMENTS

Write down all of the information you think you would need to develop a software program that would solve the following problems. Don't worry about specific procedures. List only all of the information you would need to write the software.

1.  Beach Trip – You and a friend want to drive to the beach for a weekend and you want to know (1) how long it is going to take and (2) how much the gas is going to cost.

    -   *Cost of the gas*

    -   *Number of miles to the beach*

    -   *The speed limit*

    -   *How much time it takes to stop for gas*

    -   *How much gas you have in the tank when you start the trip*

    -   *How big your gas tank is*

    -   *How many miles per gallon your car takes*

    *You also have to make certain assumptions, such as the following (others might make different assumptions, which would change the program and the answer):*

    -   *You always drive at the speed limit.*

    -   *The only time you stop is when you stop to get gas.*

    -   *When you stop for gas, you always stop for the same amount of time.*

    -   *There is no traffic.*

2.    Scheduler — You want to develop a program that automatically schedules all of your activities during the week. You want to be able to run this program every Sunday so you know the time, date, and location for each activity.

- *A list of all of the activities you are involved in for the week*

  - *Those that are flexible and can be performed on any day (e.g., working on your term paper)*
  - *Those that can only be performed at certain times (e.g., when the computer lab is available for use)*
- *How long each activity takes*

- *Whether there are any activities that need to be performed before other activities can start or finish (e.g., you have to practice your piano before your next piano lesson)*

- *What your start time is for the day*

- *What your end time is for the day*

- *If there is a deadline for any of the activities (e.g., your term paper is due on Thursday, so it is better not to schedule that work for Friday)*

- *How you want your schedule to be presented (e.g., so it looks like a calendar or just a list for each day followed by the time)*

*It might be useful to have a separate text file to hold those activities that occur every week.*

3.    The Cleaning Robot — With such busy lives these days, you decide to develop a robot that will clean up litter in a teenager's room.

*This one is a lot harder because each teenager has a different room layout and different types of litter.*

- *How often does the room need to be cleaned? This will have an impact on how much litter there is—cleaning once a month means more litter to pick up than cleaning once a week.*

- *How much litter is in a typical teenager's room when it is time to do the cleaning? This will affect the size of the bag that the robot carries to hold the litter.*

- *What distinguishes litter from nonlitter?*

- *What types of litter are there? Is the litter usually small (paper, bottles, cans, wrappers) or might it be bigger?*

- *Should the robot discriminate among articles it picks up — e. g., clothes on the floor that should go into a laundry basket as opposed to the trash can? Are there other items that should not go into the trash can? What should the robot do with them?*

- *What does a typical teenager's room look like (for example, what kind of furniture is there)? Do we need to search on top of each piece of furniture for litter or can we look just on the floor?*

- *How much time does the teenager expect (or can the teenager afford) each cleaning to take? This will have a direct impact on how fast the robot must work.*

*Specific information you would need for programming your robot includes:*

- *The amount of energy the robot needs (maybe you'll be strapping battery packs on).*

- *The size of the bag your robot will have for litter.*

- *How you plan to make your robot traverse the room. To do this, you will need a map of each room and a strategy for making sure you cover all parts.*

- *How you plan on getting around furniture.*

- *How you plan on sensing the litter (e.g., a metal detector) and the range at which your robot's sensors can sense the litter (e.g., within 1 ft., 3 ft., etc.).*

4. Several Robots — Suppose you work for United Robot Workers, Inc. (URW). Three customers approach you. Each has different needs:

   a. Customer 1, a farmer, owns a large cornfield and has trouble finding time to harvest it. She wants to know if you can provide a robot that will harvest her corn without human supervision.

   b. Customer 2 is from the Alaska National Guard, which is constantly rescuing people who wander too far afield in the tundra. Mounting a rescue party is time consuming; people have died while the members of the party are gathering. The Guard thinks having robots ready could eliminate these life-threatening delays.

   c. Customer 3, from the National Park Service, is concerned about growing amounts of litter in national parks and wants to know if you can provide a robot that can pick up the litter.

These three statements correspond to the customers' vague understandings of their problems and of potential solutions. Your task is to write a set of questions for each customer to clarify each problem.

*This exercise leads up to the laboratory in Unit 3. There, you will play the role of customer. The scope of the problem area will be restricted considerably more than it is here, making the questions easier to answer. The purpose of this exercise is to get the students thinking about robots. Questions they might pose include, but are not limited to, the following:*

- *How much is the customer willing to spend?*

- *For the Alaska National Guard, what should the robot do with the people once it finds them? Should it pick them up and carry them to safety, or should it carry shelter and supplies with it? What is an acceptable speed for the robot?*

- *How much corn (for the cornfield robot) or litter (for the National Park Service) should the robot be able to carry?*

*Remember to make students focus on requirements rather than solutions. They should not ask questions like, "What type of locomotion mechanism do you want?" or "What is the maximum speed of the robot?" As employees of URW, they should already know the answer to such questions.*

5.   Vending machines — The Student Government Association (SGA) has funds to build a vending machine room near the central hall. The principal has agreed to let the SGA go ahead if they make provisions to keep it attractive and litter free. It is your job to define the requirements for the vending machine. What information do you need to define the requirements?

*The answers to this exercise will follow a different format to support classroom discussion and lay the foundation for a homework exercise and lead-in to a Unit 2 exercise.*

*It works well to have individuals or groups put their lists on large sheets of paper and tack them to the wall. These lists can then be used in the Unit 2 discussion of similarities and differences.*

**Class Discussion:**

*You need to know the following things:*

• *What kinds of items will be sold?*

Generate a list of possibilities with the students. The idea here is to have a variety of items. The list might include soft drinks, hot soups, school supplies, snack crackers, fruit juices, nuts and candies, sandwiches, etc.

• *In what price range should the items be?*

*(There are a lot more requirements. These are just the first two that will help determine all of the other requirements.)*

*Generate lists of possibilities for each question. Then, imagine several different vending machines, each fulfilling a different requirements set. Examples:*

   – *A sandwich machine that sells only sandwiches and chips. The sandwiches may be hot or cold.*

   – *A soda machine that sells by the can or by the cup.*

   – *A snack machine that sells nuts, crackers, candies, etc.*

   – *A hot meal machine that sells soups, TV dinners, etc.*

   – *A supplies machine that sells pencils, pens, paper, scissors, folders, etc.*

*Each one of these vending machines will have its own unique set of requirements. These requirements might include a thermometer to monitor temperature, unique display requirements, varying input support (e.g., bills as well as coins), size of output bin, etc.*

Come up with exact requirements for the particular vending machine you were assigned in class. The requirements you come up with will most likely expand beyond the requirements identified in the class discussion.

*Assign each student, or small groups of students, one of the machines discussed in class. Their assignment is to list exact requirements for their particular vending machine. The requirements they come up with will most likely expand beyond the requirements identified in the class discussion.*

## DISCUSSION

In the last unit we talked about software life-cycle processes, traditional software development, and requirements. We also introduced you to the concept of megaprogramming.

In this unit, we will go into more detail on megaprogramming.

## OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Define *domain*

- Determine whether something is or is not a domain

- Define *domain engineering*

- Define *application engineering*

- Explain how domains promote reuse

# Unit 2: Concepts of Megaprogramming

## DISCUSSION

Fact of life: All people/companies have their own unique problems to solve.

In the example in this slide, two people have two problems to solve. Both are asking for simple math functions, but balancing a checkbook won't help someone learn addition, or vice-versa.

How can they go about solving their problems?

## STUDENT INTERACTIONS

- What are the differences between the two problems?

- Suppose you run a business that provides solutions to mathematical problems. What would you do to solve the problems?

## OBJECTIVE

The students should be able to:

- Understand that people/customers have unique problems to solve, regardless of how one problem may resemble another

# People's Problems

**Problem 1**

I need something that will help me balance my checkbook.

**Problem 2**
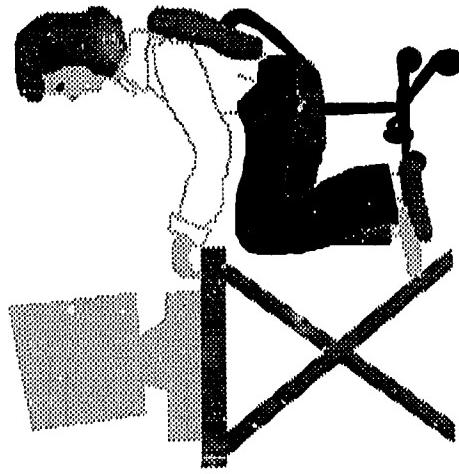
I need something that will help me teach my little sister how to add and subtract.

## DISCUSSION

You can use the following options to solve the problems on the previous slide.

- Option 1. Create two solutions, each one solving its own specific problem. For example, provide a calculator to help balance the checkbook and develop flash cards to help teach basic math skills.

- Option 2. Create a general solution that solves many problems, and try convincing your customers to adopt that solution. For example, provide a computer with multipurpose software that will solve each problem (the spreadsheet can do the math and the Pascal compiler can be used to do the flash cards). This solution, however, may be more than what each of your customers really needs or can afford.

- Option 3. First, understand the problems you are trying to solve. See if both solutions could contain some of the same components. For example, the solutions to both problems require basic math calculations. You could design calculator parts to do standard mathematical functions. You could use these parts to create 2 calculators. One would be a standard calculator that would help the first customer. The other would have an "Ask for answer" button. This would satisfy your second customer.

Doing all the up-front work required in Option 3 to understand the problems pays off in the long run because software developers typically (1) solve many problems during their career and (2) tend to specialize in an area so they end up solving problems that are similar.

How do you determine whether problems have enough in common that you should do the up-front work to see if their solutions could contain some of the same components?

## STUDENT INTERACTIONS

- In what field would you like to build software (e.g., to fly space shuttles, to build computer games)? Do you think that there are a lot of commonalities in different software programs developed in that field?

## OBJECTIVES

The students should be able to:

- Explain that there are many ways to solve problems

- Explain that doing a lot of work on understanding the customer's problems pays off in the long run

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Tr*

# Solving People's Problems

**Option 1: Generate 2 Solutions**

123.45
× / + -
C

and

Flash
Cards

---
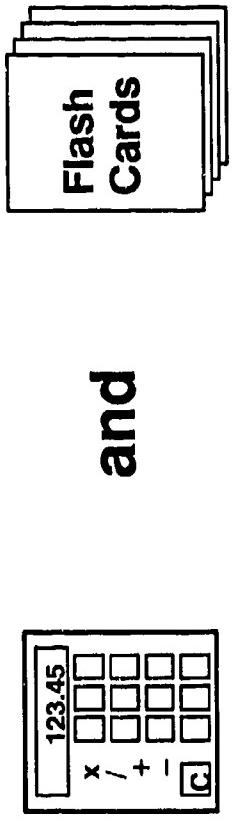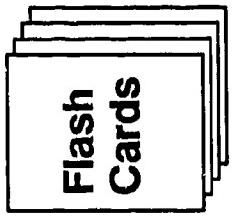
**Option 2: Generate 1 Solution that Solves Both Problems
(though it is not cost-effective)**
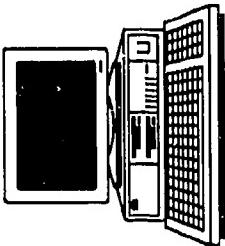
**A computer with a spreadsheet
and a Pascal compiler**

---

**Option 3: Understand How the Problems Are Related Before Generating Any Solution**

Problem 1

?
=
?

Problem 2

Overview of Megaprogramming Course:  Unit 2, Concepts of Megaprogramming. Lecture

2-3

## DISCUSSION

To have a cost-effective problem area, you need to make sure that your problems and solutions have enough in common that it makes sense to consider them together. For example, the problem of finding the area of a circle and the problem of solving linear equations do not have a lot in common — it does not pay to consider them together when creating their solutions. It is also not worth the trouble to define your domain such that thousands of problems would fit the definition: this would be too big too manage.

When there is enough in common, we can group these problems/solutions into a problem area instead of looking at each problem and solution as unique. We will now start calling these problem areas *domains*.

## Definition: *A domain* is a well-defined area where engineers speak of:

- **Common properties of problems/solutions in the area**
- **Variations among individual problems/solutions in the area**

Engineers have accumulated knowledge and skills: well thought-out laws (e.g., a robot needs to move forward and turn left to traverse any terrain) and engineering judgment (e.g., the amount of energy a robot needs to traverse a forest).

For our lab exercise, we are going to be working with the domain of robots. This domain is based on Karel the Robot, though capabilities have been added to Karel to teach you megaprogramming concepts (in fact, you will create robot programs without writing any software). The domain we have created is based on an imaginary company that builds robots. These robots perform a variety of tasks in a variety of terrains, where the basic goal is to search unknown territory for some type of object. We'll see more of Karel in the lab.

How can domains help us develop software?

## STUDENT INTERACTIONS

- Have you written procedures that you've used in a lot of your programs? Have there been procedures that would work in only one program?

## OBJECTIVES

The students should be able to:

- Understand the concept of a domain
- Be familiar with the robot domain being used for the lab exercise

# Domains

## Domain of Robots

- **Commonalities of the problems that robots solve**
  (For example, they all need to search for something.)

- **Commonalities of robot solutions**
  (For example, they all have a face-north procedure.)

- **Differences among individual robot problems and among solutions**
  (For example, some robots need special search algorithms to avoid obstacles.)

## Problem Areas

Video Games  [Robots]

Vending Machines

Cash Registers

Televisions

Minivans

Commercial Airplanes

Calculators

## DISCUSSION

Domains help us to build software that we can use repeatedly in solving many problems in a domain.

All problems in a domain are variations on a common theme, and we can expect certain similarities among them. These similarities are based on the requirements that are common to each problem in the domain.

All software programs (solutions) in a domain address similar problems. Therefore, these programs should be similar in many ways:

- Many of the procedures that make up each program

- Many of the algorithms used

- Much of the manner in which each program is tested

If we can reuse the procedures that address the similarities for all solutions in the domain, we would have to write these procedures only once.

Whenever we generate a new solution in a domain, then we have to worry about only those parts of the problem that are different from other problems in the domain.

How does megaprogramming incorporate these concepts?

## STUDENT INTERACTIONS

- Can you name other common procedures that you've used in class? What about calls to a COS function? What about calls to a READ operation that reads from a file? Do you consider this reuse? (No)

## OBJECTIVES

The students should be able to:

- Use knowledge of a domain to build software procedures that can be used to solve all problems that are in that domain

- Understand how domains support reuse

# Reuse

## Domain of Robots

- **Commonalities of the problems that robots solve**
  (For example, they all need to search for something.)

- **Commonalities of robot solutions**
  (For example, they all have a face-north procedure.)

- **Differences among individual robot problems and among solutions**
  (For example, some robots need special search algorithms to avoid obstacles.)

**Write code that solves the common parts of the problem and then reuse it in all solutions.**

(For example, the face-north procedure.)

**Write code that implements the differences among different solutions.**

(For example, the search procedure that makes sure the robot avoids obstacles.)

## DISCUSSION

Now back to our megaprogramming picture.

Let's split software development into two parts:

1. **Domain engineering,** where we:

   - Understand the problems in a domain

   - Determine the best way to create solutions to problems in that domain

   - Create reusable software

2. **Application engineering,** where we:

   - Understand the problem of a particular customer

   - Create a solution to an individual problem of a customer

   - Reuse software we've developed in domain engineering to create our solutions

Tomorrow we'll see what software development is like when we understand a domain

## STUDENT INTERACTIONS

   - Would you rather be a domain engineer or an application engineer? Why?

## OBJECTIVE

The students should be able to:

   - Understand that software development can be split into two parts:

     - Domain engineering, where we understand the problems and prepare solutions for an entire domain

     - Application engineering, where we understand a particular problem and generate a solution for that problem

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology*

# Megaprogramming

**Domain Engineering**

Understand all aspects of the domain (both problems and solutions).

→

Software and documentation to help solve a problem in the domain

Feedback

**Application Engineering**

Create a solution to a problem in the domain.

The Solution

Your Customer

Customer Requirements

*Virginia*
**CENTER of EXCELLENCE**
*for Software Reuse and Technology Transfer*

2-6

# UNIT 2: CONCEPTS OF MEGAPROGRAMMING

## SUMMARY

### DOMAINS

- Domains contain related problems and solutions that have:

    - Similarities among problems

    - Solutions with common parts

    - Variations among the problems and solutions



```
Problem Areas

Video Games          Robots

Vending Machines

            Cash Registers
Televisions
        Minivans

Commercial Airplanes

        Calculators
```

```
Domain of Robots

● Commonalities of the
  problems that robots solve
  (For example, they all need to
  search for something.)

● Commonalities of robot
  solutions
  (For example, they all have a
  face-north procedure.)

● Differences among
  individual robot problems
  and among solutions
  (For example, some robots
  need special search
  algorithms to avoid
  obstacles.)
```

- When defining domains:

    - Make sure the problems and solutions have enough in common that it pays to consider them together.

    - Do not include large numbers of barely-related problems in the same domain.

- When identifying a problem in the domain, you only need to identify how it differs from other problems. What is common to all problems defines the other characteristics of the problem.

- When solving a problem in the domain, you can make use of what is common to all solutions.

**Domain of Robots**

- **Commonalities of the problems that robots solve**
  (For example, they all need to search for something.)

- **Commonalities of robot solutions**
  (For example, they all have a face-north procedure.)

- **Differences among individual robot problems and among solutions**
  (For example, some robots need special search algorithms that will allow them to avoid obstacles.)

**Write code that solves the common parts of the problem and then reuse it in all solutions.**
(For example, the face-north procedure.)

**Write code that implements the differences among different solutions.**
(For example, the search procedure that makes sure the robot avoids obstacles.)

MEGAPROGRAMMING

Megaprogramming has two main tasks:

1. **Domain engineering,** where we:

   - Understand the problems in a domain

   - Determine the best way to create solutions to problems in that domain

   - Create software that is reusable in all solutions in the domain

2. **Application engineering,** where we:

   - Understand the problem of a particular customer

   - Create a solution to an individual problem of a customer

   - Reuse software we have developed in domain engineering to create our solutions

# UNIT 2: CONCEPTS OF MEGAPROGRAMMING

## EXERCISES

1. Continue with your vending machine problem (Unit 1, Problem 5). On the board, or on large sheets of paper, list the requirements generated by the students. Ask the following questions:

**Similarities:**

- Are there any similarities among the requirements for the different vending machines?

- Could a manufacturer design a component for each similarity?

**Differences:**

- What requirements are different from vending machine to vending machine?

- How could the differences be accommodated? Could any of the differences be a simple modification of an already identified component? Would it be necessary to build an entirely new component?

Based on these components, what components do you need to come up with for your vending machine? This could include similar components as well as components that are different from all other vending machines. You should also identify which components need to interact with each other, which components you feel are reusable across other vending machines, and which components are unique to your vending machine.

Each group of students working on a particular vending machine should come up with a list of components that they need to build that vending machine. Each group should present its final list of vending machine components to the class. For each vending machine, discuss the following questions:

- Have they designed a vending machine?

- Were they able to identify "reusable" components (i.e., components that could be used with little or no modification)?

- What components did they have to create to handle requirements unique to their vending machine?

- Would they consider vending machines a class of common problems and solutions (a domain)?

- What are some of the benefits of going through these steps?

When you are finished, answer the following questions:

- Could you use megaprogramming concepts to help build vending machines?

- What would the domain engineer do in this domain?

- What would an application engineer do in this domain?

2. Discuss the robot problem from Unit 1, Problem 4.

- Make a list of common jobs and tasks that the three robots in Unit 1, Problem 4 needed.

- Make a list of specific jobs and tasks that not all the robots needed.

## HOMEWORK

1. Consider the following—are they domains? Why or why not?

- The process of applying to college

- The process of proving equations

- The process of school bus scheduling

- The process of transportation scheduling

2. Describe a domain in today's world of teenagers. List the similarities and differences in your domain.

# UNIT 2: CONCEPTS OF MEGAPROGRAMMING

## TEACHER NOTES FOR EXERCISES

1. Continue with your vending machine problem (Unit 1, Problem 5). On the board, or on large sheets of paper, list the requirements generated by the students. Ask the following questions:

**Similarities:**

- Are there any similarities among the requirements for the different vending machines?

  *Examples of similarities might include the need for the following: temperature monitor, display, input, output, storage modules, etc.*

- Could a manufacturer design a component for each similarity?

  *This should generate a list such as coin boxes, mechanisms to deliver the merchandise, display units, utilities units, storage units, housing units.*

**Differences:**

- What requirements are different from vending machine to vending machine?

  *Examples of differences might include a microwave to heat an item, a special option that makes change for bills, etc.*

- How could the differences be accommodated? Could any of the differences be a simple modification of an already identified component? Would it be necessary to build an entirely new component?

  *For example: A microwave to heat an item would probably have to be a new component. A bill changer could probably be a modification of the existing coin/bill input mechanism.*

Based on these components, what components do you need to come up with for your vending machine? This could include similar components as well as components that are different from all other vending machines. You should also identify which components need to interact with each other, which components you feel are reusable across other vending machines, and which components are unique to your vending machine.

*Assign a group of students to each of the vending machines identified in the Unit 1 exercise. Based on the components discussed today, have them identify what components they will need to come up with for a complete vending machine. This could include similar components as well as components that are different from all other vending machines. They should also identify which components need to interact with each other, which components they feel are reusable across other vending machines, and which components are unique to this vending machine.*

*This can be done either as a homework assignment or as a small-group exercise at the end of Unit 2 or before Unit 3.*

Each group of students working on a particular vending machine should come up with a list of components that they need to build that vending machine. Each group should present its final list of vending machine components to the class. For each vending machine, discuss the following questions:

- Have they designed a vending machine?

  *See if the other students can identify any missing components. The point of this question is to help the students see that, like requirements, making sure that you have everything is difficult.*

- Were you able to identify "reusable" components (i.e., components that could be used with little or no modification)?

  *The students should understand why having reusable components can save time and money. These components can be software programs or actual vending machine hardware components: the idea of savings remains the same.*

- What components did they have to create to handle requirements unique to their vending machine?

  *All solutions will have unique parts. If there were no unique parts, then the solution would be exactly identical to another problem/solution and you would only have to build one solution.*

- Would they consider vending machines a class of common problems and solutions (a domain)?

  *Yes. There are enough similarities to make it worth your while to understand the similarities and differences among vending machines and to make use of that knowledge each time you build a new one.*

- What are some of the benefits of this procedure?

  *Savings in design, savings in manufacturing, aesthetic uniformity, etc.*

When you are finished, answer the following questions:

- Could you use megaprogramming concepts to help build vending machines?

  *Yes. There is enough in common between vending machines, yet enough differences, that it makes sense to study their similarities and differences.*

- What would the domain engineer do in this domain?

  *The domain engineer would create reusable vending machine components and documents that describe how to use those components to build vending machines.*

- What would an application engineer do in this domain?

  *An application engineer would talk to a customer and use the products created by the domain engineers to define and validate requirements that met the customer's need and build a vending machine that satisfied those requirements.*

2. Discuss the robot problem from Unit 1, Problem 4.

   - Make a list of common jobs and tasks that the three robots in Unit 1, Problem 4 needed.

   - Make a list of specific jobs and tasks that not all the robots needed.

     *The answer to these two questions depends on the students' answers to Problem 4 in Unit 1. However, they might observe that all the robots move, and they search for some type of object. The type of object, and the robot's response to finding it, are two things that vary among the three robots.*

## TEACHER NOTES FOR HOMEWORK

1. Consider the following—are they domains? Why or why not?

   - The process of applying to college

     *Yes. Colleges usually ask for many similar types of information on their application forms, yet there are enough differences that you could not use the same application at more than one school without any changes.*

   - The process of proving equations

     *Yes. You follow similar steps in solving any equation. However, the order in which you follow the steps and the exact steps you follow will vary from equation to equation.*

   - The process of school bus scheduling

     *Yes. School bus scheduling will have the same coordination and logistics problems from school to school and county to county. However, there will be enough differences (e.g., number of buses, size of district, etc.) that you could not use the same school bus scheduling system for every school.*

   - The process of transportation scheduling

     *No. This domain would be too large to justify establishing a domain. There are similarities between different types of transportation; however, there are too many differences from one transportation type to another and not enough similarities that it will not pay to generate and use the domain.*

2. Describe a domain in today's world of teenagers. List the similarities and differences in your domain.

   - *The answers for this question will vary. Look for a domain that has enough similarities between the problems and solutions and significant differences that it would make sense to establish and use a domain whenever you need to generate a solution.*

*This page intentionally left blank.*

## DISCUSSION

Unit 2 introduced domains and the importance of considering problems and solutions in the context of a domain.

In Unit 3, we'll see how domains can simplify software development. We'll concentrate on our robot domain and show a straightforward, step-by-step process for developing robot software. In the laboratory, you'll get the chance to use this process.

## OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Understand that a problem can be defined solely in terms of its differences from other problems in its domain

- Understand that an application can be developed by following a well-defined process that involves:

  - Requirements definition

  - Validation

  - Solution generation from precisely-defined requirements

# Unit 3: Application Engineering

## DISCUSSION

This slide ties concepts from Unit 2 about problems versus solutions to a software development process for defining problems and creating solutions to those problems.

The prototypical steps in developing software—using megaprogramming or otherwise—are as follows. Customers start with a vague understanding of an existing problem and a realization of the need for a solution to that problem (symbolized by the outline of the robot).

- Step 1: The application engineer states a customer's problem precisely (the vision of the complete robot). The ability to state the problem precisely is a measure of how well the problem is understood. The application engineer's knowledge about the problem area helps in reasoning about what robot the customer needs (hence the arrows both from and to the application engineer).

- Step 2: The application engineer generates a solution to the problem based on the problem statement created for the customer (the solution is symbolized by the robot).

This slide does not show what happens after the software is developed, although slides in earlier units did.

Ideally, we as application engineers should always think in terms of problems and solutions as we develop software. We should keep these two parts separate in our minds.

**A precise statement of a problem defines the requirements for a solution.** As we saw in Unit 1, it's important that we be precise when we define requirements.

What do these two activities (precisely stating the problem and generating a solution) really mean? Let's explore this question, in terms of megaprogramming.

## STUDENT INTERACTIONS

- For the vending machine exercise, did you precisely state the problem? Did you miss any parts of the problem? Could somebody generate a vending machine from your list of requirements?

## OBJECTIVES

The students should be able to:

- Explain the notion of software development as a two-step process: precisely stating the problem and generating the solution

- Explain why requirements are the outcome of precisely stating a problem

# Application Engineering



Step 1:
Precisely
State
Problem

Knowledge

Step 2:
Generate
Solution

Application
Engineer
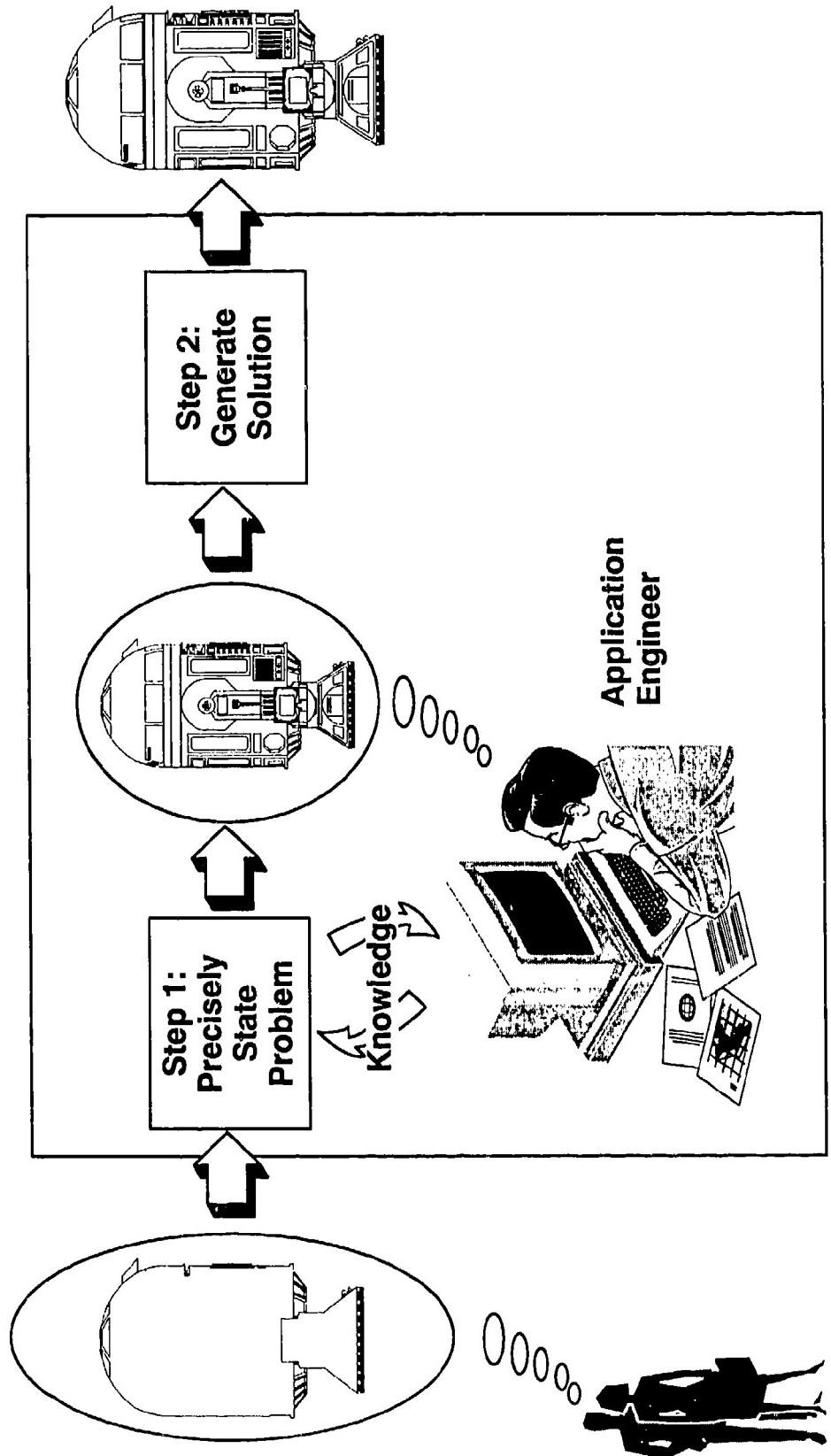
## DISCUSSION

This slide illustrates how we can precisely state a problem in the context of a domain, once we have analyzed all problems in a domain to the point where we understand the similarities and differences among them.

In general, precisely stating problems so that other people can understand what we mean is hard. We must be unambiguous. We cannot assume people understand anything we do not explicitly state. However, precisely stating a problem is much easier if we know people share certain assumptions about the problem—that is, the domain of which it is a part. These assumptions can be:

- Things we know to be common in all problems in the domain. For example, all people who buy automobiles have a transportation problem that they assume a car will solve.

- Things we know to vary across problems in the domain. Some people who buy automobiles haul cargo. Others haul families. Still others want a sports car. No single car is ideal for all three purposes.

If we really understand a domain, we can state a problem strictly in terms of variations among problems. A decision tree is one way to do this (note that what's on the slide is compressed; it's not actually a tree). For cars, a decision tree can lead us to identify a particular car with specific options. Each tree level represents a statement about your problem that will lead to a different solution. The first level tells you whether you should buy a pick-up truck, a sedan, or a convertible. The second level tells you if you need four-wheel drive. The third tells you if you need air conditioning, etc. (The order among the levels is arbitrary.)

This is pretty much how you decide what car to buy. It works because you and the auto dealer both know what an automobile is. You share a common vocabulary ("coupe" versus "hatchback"). Suppose you didn't share that vocabulary—how would you explain what you wanted? You would have to describe everything (number of wheels, horsepower, number of seats, etc.). But you do share a vocabulary, so you have to only describe how your transportation needs differ from those of other people.

When you buy an automobile, the salesperson acts as an application engineer. (This analogy isn't perfect, because an application engineer typically knows more about the domain and product than the customer.)

## STUDENT INTERACTIONS

- Name another common situation where you make assumptions about what other people know. (For example, when you purchase a stereo, you assume it will have a volume control.)

## OBJECTIVES

The students should be able to:

- Explain that, within a domain, one problem can be defined in terms of its differences from other problems
- Explain a few of the commonalities and variations for the automobile domain

# Precisely Stating Problems

Automobiles

- haul cargo
- haul family
- haul self

- drive on dirt
- drive on pavement

- live in a cold climate
- live in a hot climate
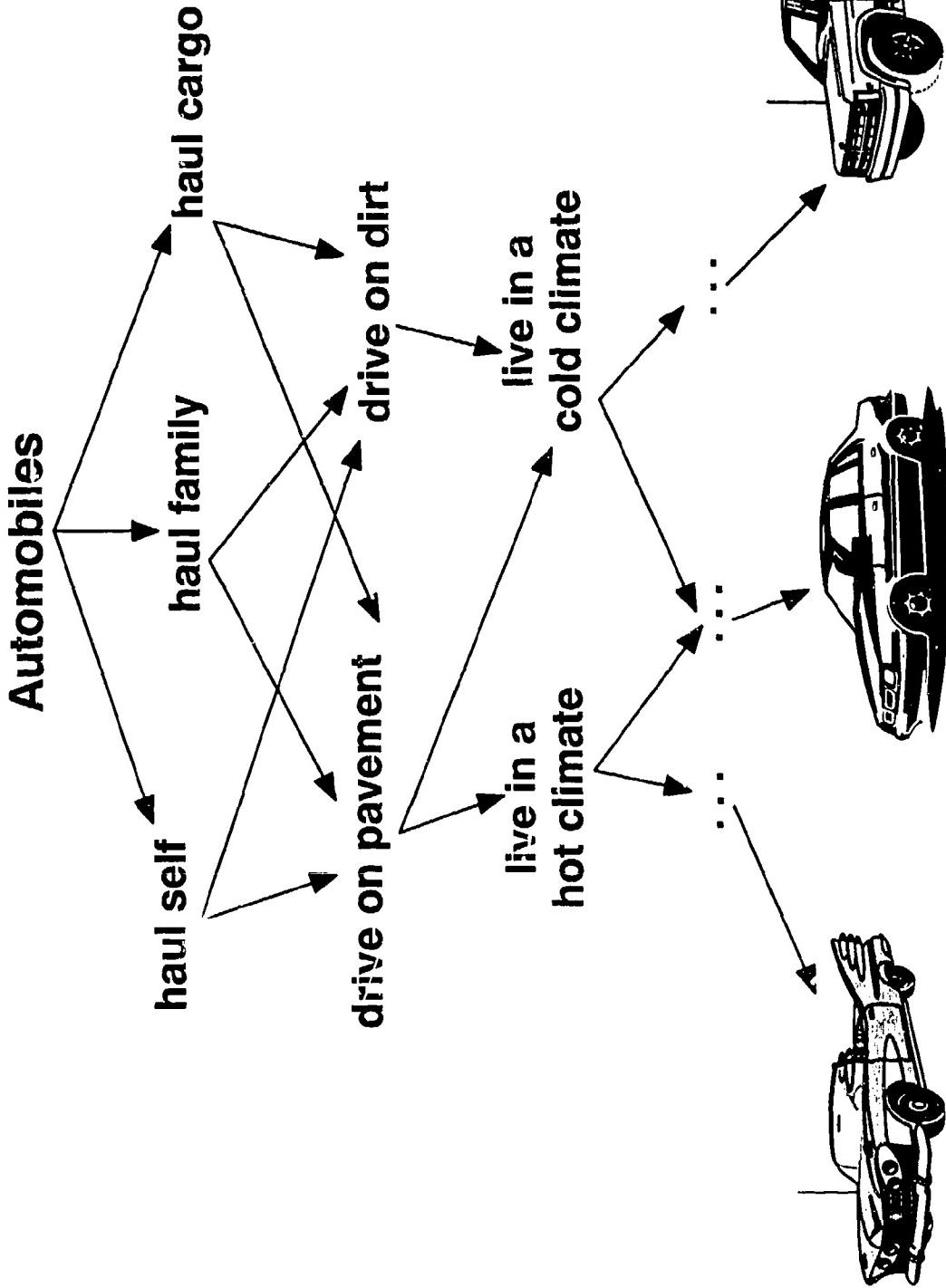
## DISCUSSION

We can study domains other than automobiles, too. Domains of software can be described in terms of commonalities and differences. Consider our robot domain. The following are examples of characteristics that all robots in the domain share:

- All robots move.

- All robots search for some type of object.

- All robots are autonomous: once started, they perform their mission without human intervention.

Similarly, we can identify how a robot differs from other robots in the domain:

- The type of terrain in which it operates

- The type of object for which it searches

- The type of mission it performs

Unless you understand what's common to all robots in the domain, these differences do not make much sense. But interpreted in the context of things common to all problems in a domain, these differences are enough to precisely state a single problem (and so to define the requirements for a solution to that problem).

(As with Slide 3-3, space limitations necessitated a compressed decision tree. A leaf does not uniquely identify a set of decisions, although following a path from the root to a leaf does.)

Is there a process we can follow to precisely state the problem?

## STUDENT INTERACTIONS

- What would a decision tree look like for buying a stereo?

## OBJECTIVES

The students should be able to:

- Explain that a particular problem in a domain can be defined in terms of how it differs from other problems in the domain

- Explain a few of the commonalities and variations for the robot domain

# Precisely Stating Problems (cont.)

Robots
├── search a forest
├── search a tundra
└── search a building ... 

search a forest → sparse, dense, average

... mineral deposit

surveillance equipment

sparse, dense, average → person, litter

person → don't pick up objects ...

litter → pick up objects ...
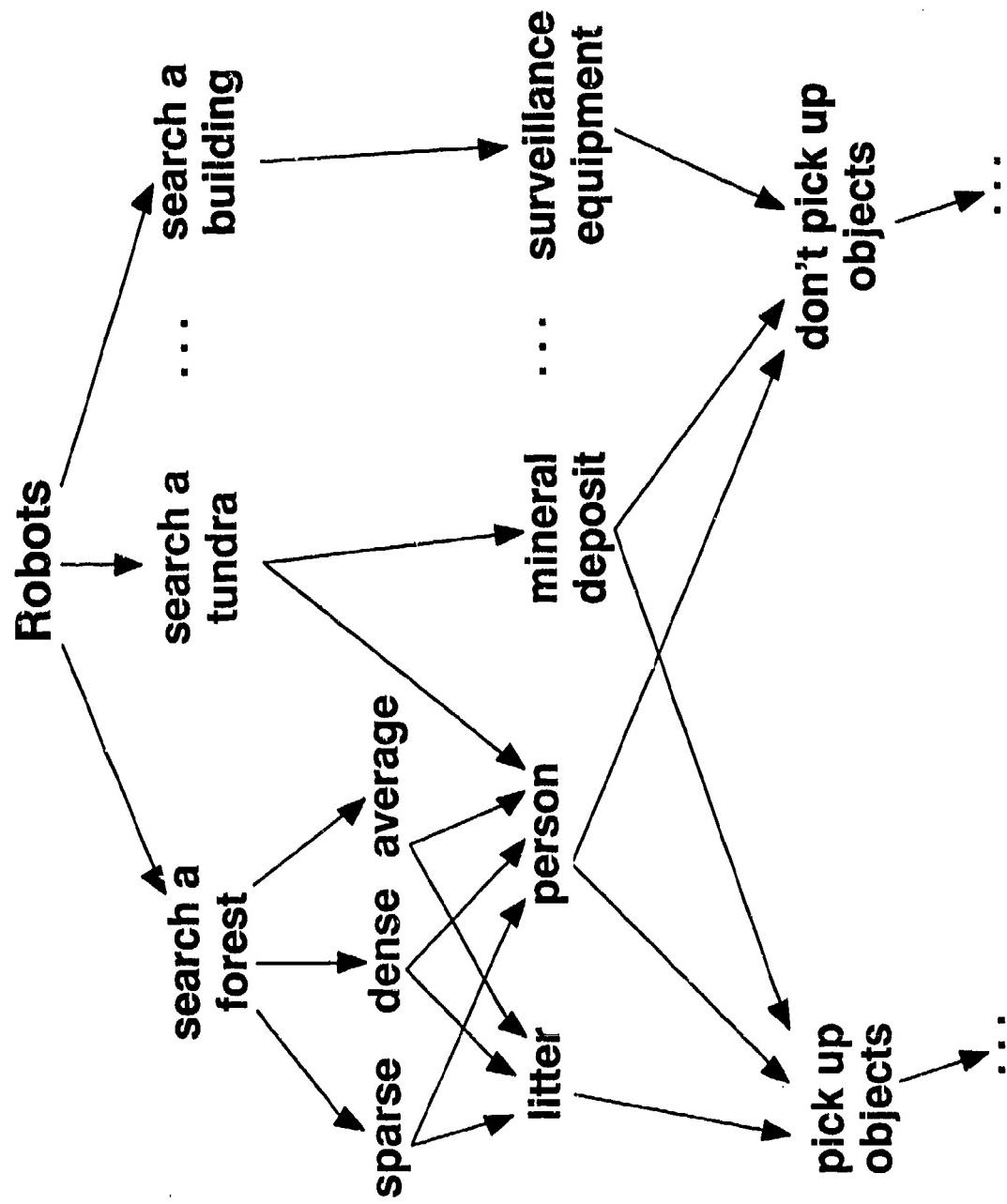
## DISCUSSION

This is a process for creating requirements for robots in our robot domain. Just as with automobiles, we have reduced creating robots to making a finite number of decisions:

1. In what type of terrain will the robot operate?
2. How should it search the terrain?
3. If it's searching a forest, how dense is the forest?
4. For what type of objects will it search?
5. Should it pick up objects as it finds them?
6. Where should it end up?
7. How many objects should it be able to carry?
8. How many batteries will it need?

The application engineer works with customers to resolve these decisions. The result is a complete set of requirements. Note that you can't make sense of a set of decisions without knowing what they mean in the context of a domain. For instance, "forest, sweep, average, litter, yes, its origin, 200, 25" is a valid set of answers for the above questions. But you can't really tell what that means unless you understand the commonalities.

The decisions influence each other:

- Robots that operate in a field don't search for the same types of objects as robots that operate in a tundra.
- Robots whose mission is to search for objects but not pick them up don't need any carrying capacity.
- People familiar with the domain know the best searching strategy (sweep or zigzag), except for robots that operate in a forest.

In other words, the answer given for one decision may affect the range of answers that are valid for subsequent decisions, or even whether the decision must be made. For this reason, the decisions are ordered into a process. How does the application engineer find out the answers to these decisions? The customer always knows the answers to some (e.g., terrain). The customer may not know the answers to others (e.g., number of batteries). The application engineer must use "engineering judgment." How many objects is the robot expected to carry on an average mission? How heavy are they? This will affect the number of batteries needed. How do customers know if the requirements really meet their initial vague understanding of the problem?

## STUDENT INTERACTIONS

- When buying a stereo, does it matter how the decisions are ordered?

## OBJECTIVE

The students should be able to:

- Explain the process for creating robot requirements

# Precisely Stating Problem for a Robot

**Step 1: Precisely State Problem**

1. Choose Terrain Type
2. Choose Search Strategy
3. Choose Forest Density
4. Choose Object Type
5. Choose if Objects are to be Carried
6. Choose Ending Position
7. Choose Carrying Capacity
8. Choose Number of Batteries

Robot Requirements

**Step 2: Generate Solution**

Customers

Application Engineer

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Transfer*

## DISCUSSION

This slide introduces validation into the software process.

Before generating a solution, the application engineer needs to know if the requirements really capture the customer's problem. It's important to do this early, while fixing errors is relatively inexpensive (see Slide 1-6).

We enlarge our view of the process for precisely stating a problem to include validating requirements.

The application engineer makes sure the requirements really express the behavior the customer intended. For instance, does the robot really perform the mission the customer needs? Does the robot have enough batteries to perform its mission in the forest where its customer wants to use it?

As before, the result of the whole "Precisely State Problem" step is the robot requirements. This slide shows that the requirements have an intermediate, nonvalidated form.

Validating requirements is done by a variety of techniques. Simulation and analytical models are often used. For example, application engineers might calculate the number of batteries by using an equation that describes a robot's battery consumption to calculate whether the robot can perform its mission.

If the application engineer's statement of the problem is not valid, the application engineer needs to rethink certain decisions (the arrow leading back to the decision-making portion of the process).

Okay, the customer thinks the requirements are correct. What's the best way for the application engineer to generate a solution?

## STUDENT INTERACTIONS

- Can you imagine how a customer paying $10 million for development of a software program feels when, upon receipt, they find that it wasn't really what they wanted? THIS HAPPENS! What would you do? Validating requirements will help that problem.

## OBJECTIVE

The students should be able to.

- Understand the role of validating requirements in application engineering

# Validating Requirements

**Step 1: Precisely State Problem**

3-6
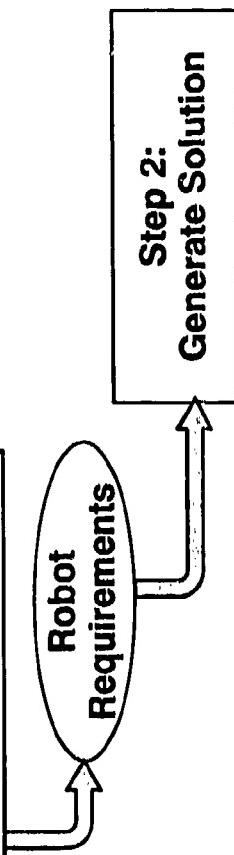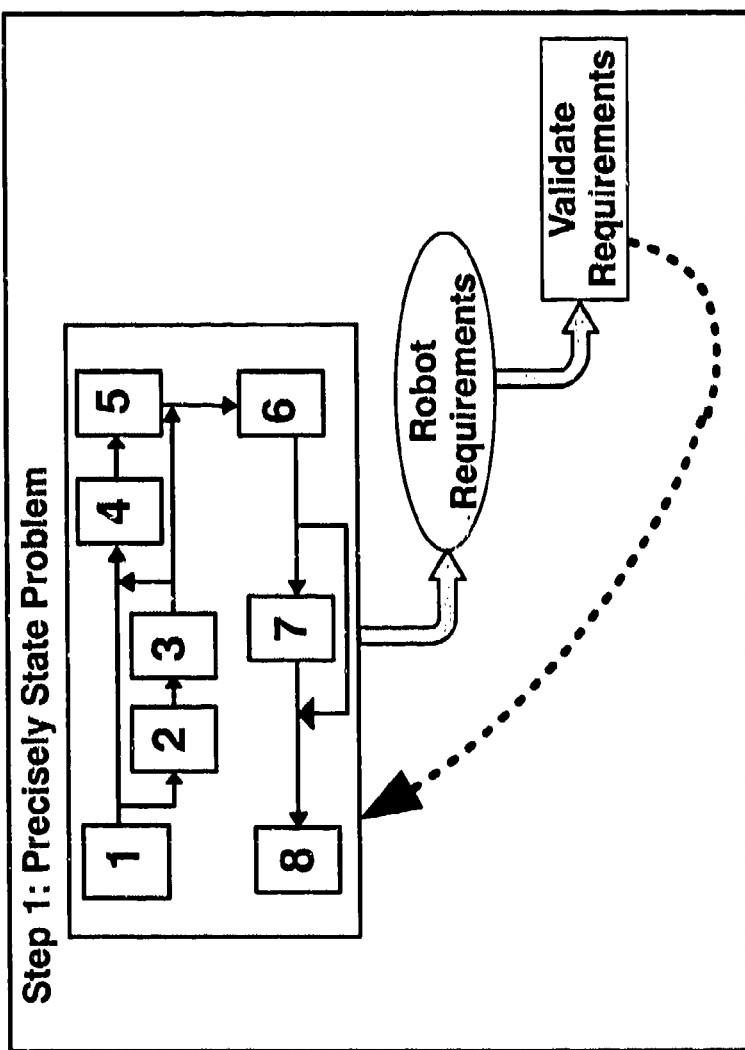
## DISCUSSION

This slide shows the main concepts in generating solutions from requirements. It also shows how existing software can be reused.

Assume the application engineer's company understands the robot domain, having built some robots for previous customers. That means old parts should be available (from domain engineering) to build robots.

Think of the robot as consisting of two types of parts: (1) parts needed by all robots in the domain plus (2) parts needed by the particular robot the customer needs.

Since parts are available, the application engineer can build a robot by:

1. Selecting the necessary parts (for instance, all robots need an object sensor and a compass; a robot in a tundra always needs a rock-avoiding algorithm, but a robot in a cornfield doesn't).

2. Composing them to create a robot. This works for both hardware parts and software parts!

The requirements are precise enough to identify a particular robot; that means they are also precise enough to identify the set of parts the application engineer needs to build that robot.

Once the application engineer composes those parts, the robot is built.

Ideally, the application engineer won't even need to build any parts—if the right previously built parts are available. (If the customer wants a robot with features that haven't been built before, the application engineer will need to create them.) Also, if someone could figure out in advance which parts are needed to solve which problems, selection and composition could be completely automated. This is possible—it's the role of domain engineering, as we'll see in Unit 4. In the laboratory for Unit 3, we'll see examples of using domain engineering products to build robot software automatically.

## STUDENT INTERACTIONS

- How would this process work for the vending machine domain? for the automobile domain?

## OBJECTIVE

The students should be able to:

- Explain the purpose of the second step of application engineering in developing software: Generating a Solution

# Generating a Solution
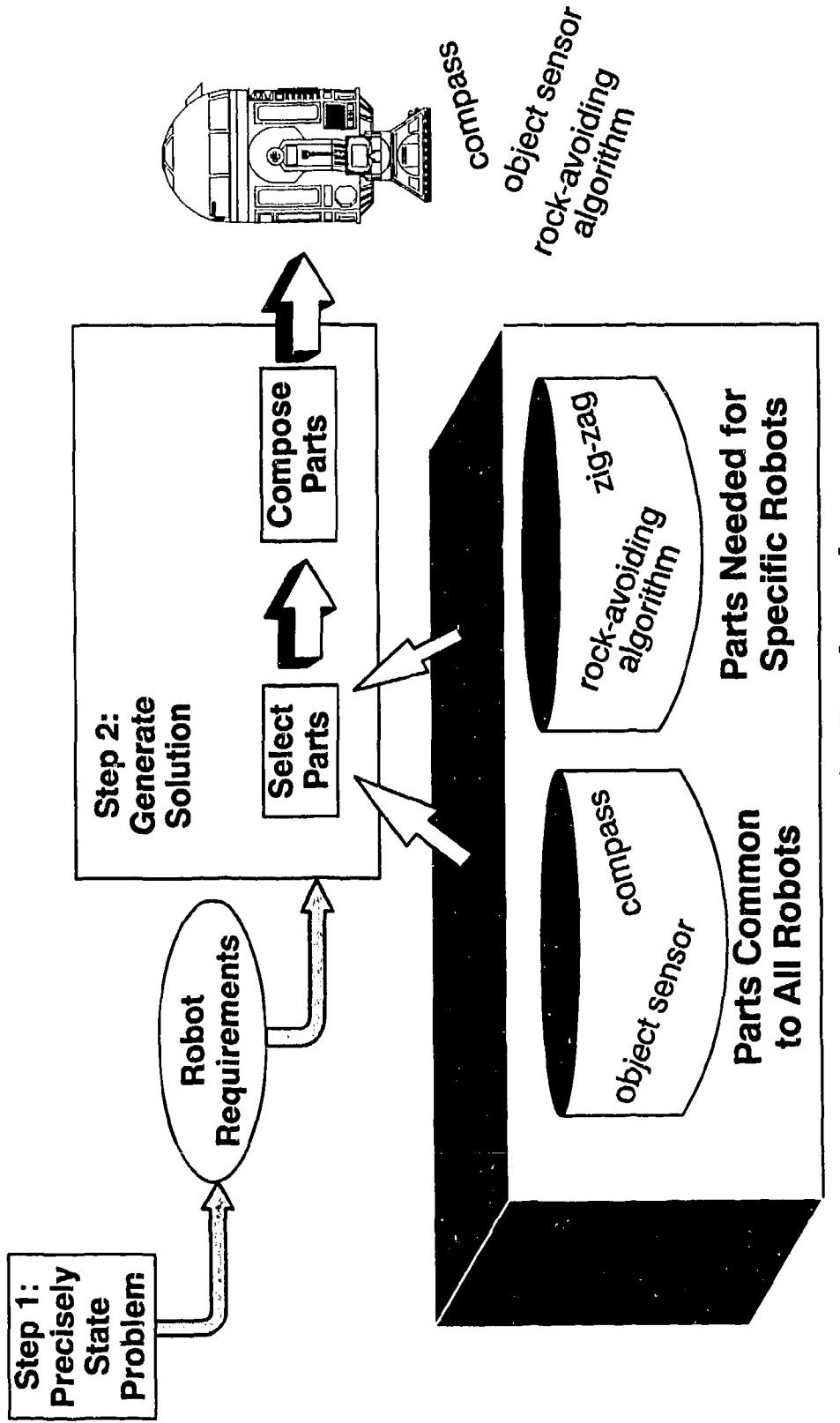
**Step 1: Precisely State Problem**

Robot Requirements

**Step 2: Generate Solution**

Select Parts

Compose Parts

compass
object sensor
rock-avoiding algorithm

Parts Common to All Robots

object sensor
compass

Parts Needed for Specific Robots

rock-avoiding algorithm
zig-zag

**Parts from Domain Engineering**

# UNIT 3: APPLICATION ENGINEERING

## SUMMARY

Application engineering involves:

- A customer who has a problem

- An application engineer who solves the problem

An application engineer solves the problem by:

1. Understanding and precisely stating the problem AND

2. Generating a solution based on the problem statement

**Customer's Problem Statement** ⟹ **Step 1: Precisely State Problem** ⟹ **Step 2: Generate Solution** ⟹ **Solution**
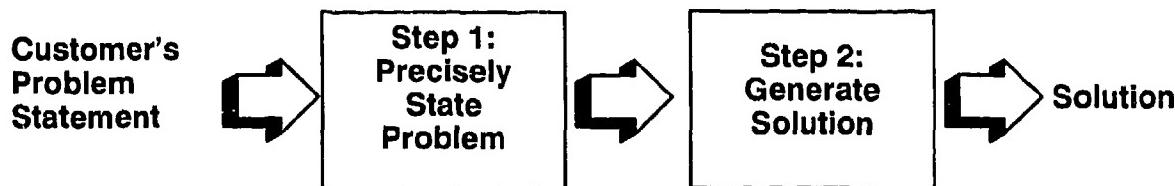
### STEP 1: PRECISELY STATE PROBLEM

To understand a problem, it is easier if the application engineer understands other related problems:

- What the problem has in common with other, similar problems

- How the problem differs from these other, similar problems

An application engineer precisely states the problem in terms of the domain by:

- Deciding how the problem differs from other problems in the domain. These decisions will require engineering judgment in addition to cold, hard facts.

- Validating the problem statement (i.e., the requirements) to make sure they precisely express the behavior the customer intended.

The following decision trees show part of the decisions needed to identify how problems differ in the automobile and robot domains.

## Example Decision Trees for Precisely Stating the Problem

**Automobiles**

haul cargo    haul families    haul self

drive on pavement    drive on dirt

live in a hot climate    live in a cold climate

. . .    . . .    . . .

**Robots**

search a forest    search a tundra    search a building

sparse   dense   average

litter    person    mineral deposit    surveillance equipment

pick up objects    don't pick up objects

. . .    . . .
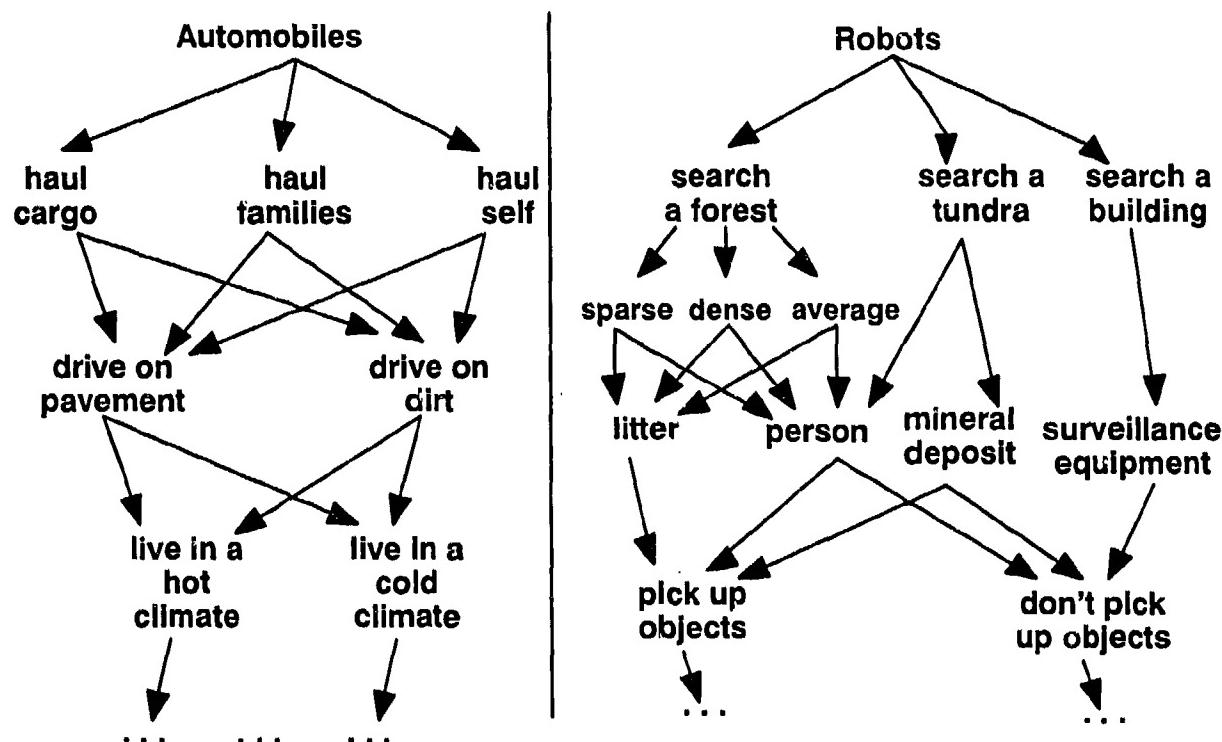
STEP 2: GENERATE SOLUTION

The application engineer then generates a solution based on the precise problem statement from Step 1. To do this, the application engineer uses the application engineering environment set up by the domain engineer. This environment contains:

- Software components needed to generate a solution to a problem in the domain. These components include:

    - Components that are common to all solutions

    - Components that solve only specific problems

- Help for how to put all of these components together to form a solution.

The following figure represents what happens in the two steps of application engineering.

# APPLICATION ENGINEERING:

## Customer's Problem Statement

## Step 1: Precisely State Problem

**Precisely State Problem**

Robot Requirements

Validate Requirements

## Step 2: Generate Solution

Select Parts

Compose Parts

Common Parts

Specific Parts

Parts from Domain Engineering
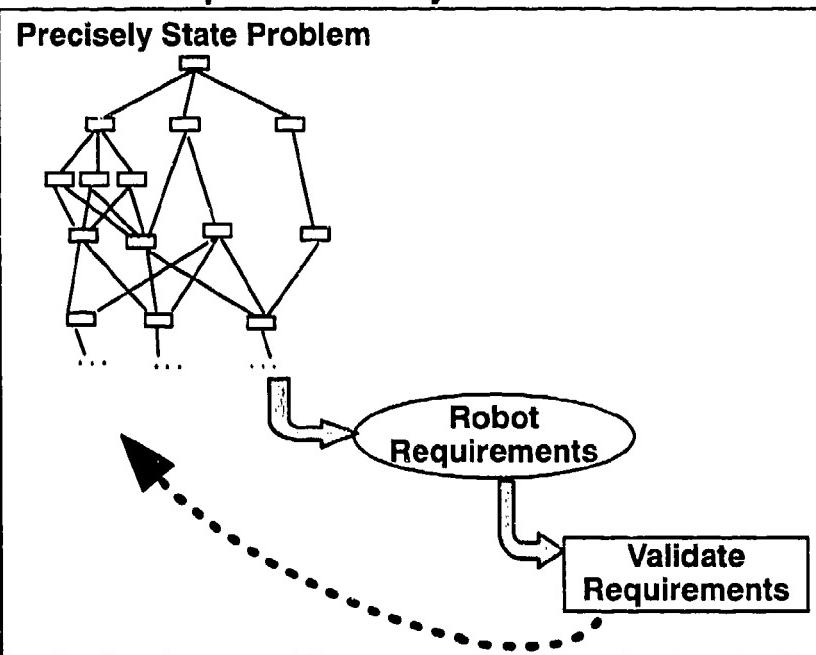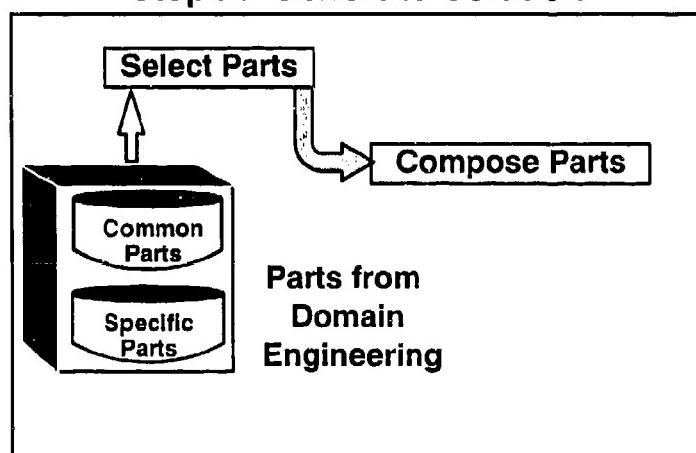
## Solution

*This page intentionally left blank.*

# UNIT 3: APPLICATION ENGINEERING LABORATORY

## PART 1: BACKGROUND

In this laboratory, you will practice application engineering. Imagine yourself to be an application engineer who works for URW. Three customers approach you. Each has different needs:

1. Customer 1, a farmer, owns a large cornfield and has trouble finding time to harvest it. She wants to know if you can provide a robot that will harvest her corn without human supervision.

2. Customer 2 is from the Alaska National Guard, which is constantly rescuing people who wander too far afield in the tundra. Mounting a rescue party is time-consuming; people have died while the members of the party were gathering. The Guard thinks having robots ready could eliminate these life-threatening delays.

3. Customer 3, from the National Park Service, is concerned about growing amounts of litter in national parks, and wants to know if you can provide a robot that can pick up the litter.

These three statements correspond to customers' vague understandings of their problems and of potential solutions. Your task in this laboratory is to help these customers understand their problems fully and to provide them with robots that solve their problems. To assist you in this, we have provided you with a tool that automates some of the application engineering. Part 2 describes its use.

In brief, you will be asked to generate the software for a robot. You will do so by following the application engineering process for precisely stating a problem, which you saw in class. Some of the decisions you must make can be answered from the three statements above. Others may require clarification from your customer. Your instructor will act as the customer, answering questions you might have on the requirements for the robot. Keep in mind, though, that a customer does not necessarily know everything. As an application engineer, you are expected to use your own expert judgment when your customer does not know what choice is right.

## PART 2: EXERCISES

### 1. CORNFIELD ROBOT

URW manufactures robots that can harvest corn. You must act as an application engineer and help solve your customer's problem by resolving the decisions in the domain. By doing so, you will create a model of a robot that harvests corn. You can use this model to generate the software that controls the robot. However, you cannot just generate any corn-harvesting robot. In the first place, your customer has a specific requirement: she wants the robot to end its mission at its point of origin. In the second place, she cannot spend more than $13,500.00. The robot you model must not exceed this price. Better still, it must be the least expensive robot that can do the job.

You will be informed of the robot's price as part of validation. However, you should know that two factors determine a corn-harvesting robot's price. The first factor is the maximum number of ears of corn it can carry. URW offers its customers robots that carry between 50 and 500 ears, in multiples

of 10. (The decision to carry 53 ears therefore results in a robot that costs the same as one that carries 60 ears, although the former robot will still pick up, at most, 53 ears.)

The second factor is the number of batteries with which the robot is equipped. All robots have at least one battery. Each extra battery costs money, but increases the distance the robot can travel. To estimate the **minimum** number of batteries needed, click on the Help button when you are asked to make the decision on the number of batteries.

Question A. Find the most appropriate robot for your customer. Do so by repeating the process of precisely stating the problem, varying the decisions until you believe your model of robot is right. Use the following graph to correlate the cost of each robot to its carrying capacity. What trend do you observe?
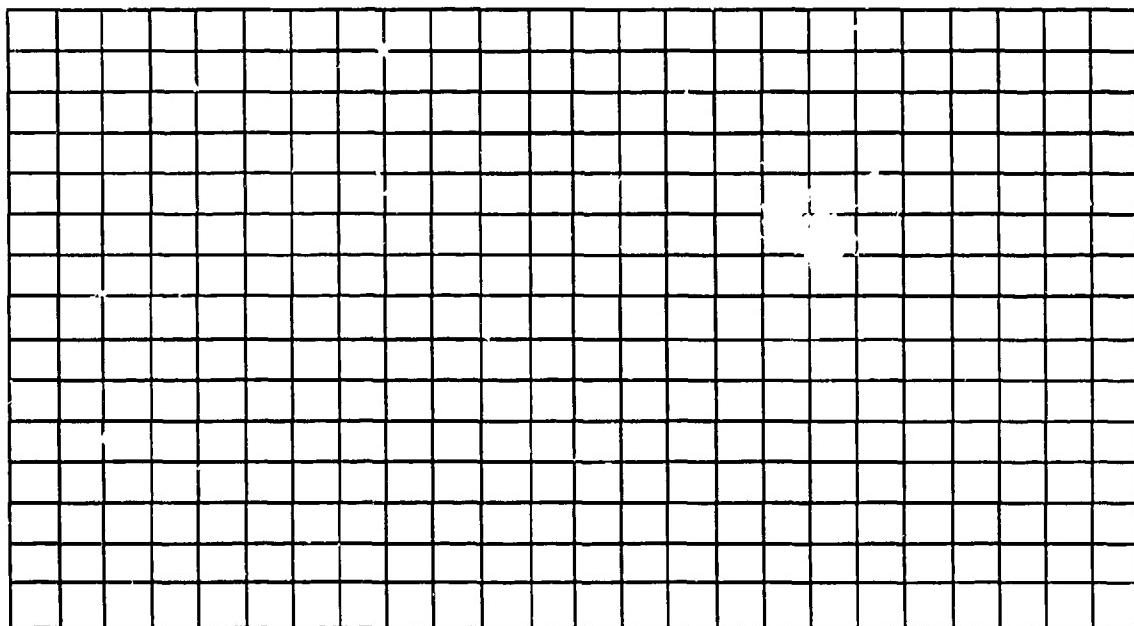


**Carrying Capacity of Robot**

Question B. Generate and execute the software for three robots: the one you described in Part A, the one with the minimum carrying capacity, and the one with the maximum carrying capacity. Record the time needed for each one to execute. Which takes the least time? Would you have created a different robot for your customer if time for harvesting had been her highest priority?

## 2. RESCUING ROBOT

Generate a robot that meets the needs of your second customer. What decisions related to choosing the mission are clearly invalid? Why?

Run your robot several times. Notice that there is more than one tundra for your robot to search. Compare their characteristics. Which one requires more energy? Would you recommend that your customer equip his robot with enough batteries to handle either case, or do you think your customer would be satisfied with a less expensive robot that could only handle the low-energy case?

## 3. LITTER-GATHERING ROBOT

Generate a robot that meets the needs of your third customer. For this customer, you will find that you need to try more than one robot to determine which one is best. The reason is that URW has two searching strategies for robots that operate in a forest. One is to "sweep" back and forth, horizontally; the other is to zigzag. Which is more effective depends on the forest in which the robot is operating.

Precisely state the problem for this robot. Use the results to fill in the following table:

| Density of Trees in Forest | Cost of Robot | |
|---|---|---|
| | Search by Sweeping | Search by Zigzagging |
| sparse | | |
| average | | |
| dense | | |

What do you observe about robot cost versus forest density?

## PART 3: USING THE APPLICATION ENGINEERING ENVIRONMENT

This part of the laboratory describes how to use the application engineering environment for specifying and generating robot software. Your instructor will tell you how to invoke the environment. Once you have done so, you will see the following dialog box (the main dialog box):

```
╔══════════ Application Engineering Environment ══════════╗
║                                                         ║
║            Application Engineering Environment          ║
║                           For                           ║
║                       Robot Domain                      ║
║                                                         ║
║     Select a process step:                              ║
║                                                         ║
║                  ┌─────────────────────────┐            ║
║                  │  Precisely State Problem │            ║
║                  └─────────────────────────┘            ║
║                  ┌─────────────────────────┐            ║
║                  │    Generate Solution     │            ║
║                  └─────────────────────────┘            ║
║                  ┌─────────────────────────┐            ║
║                  │ Execute and View Solution│            ║
║                  └─────────────────────────┘            ║
║                                                         ║
║                                                         ║
║   ┌─────────────────┐         ┌─────────────────┐       ║
║   │      Quit       │         │      Help       │       ║
║   └─────────────────┘         └─────────────────┘       ║
║                                                         ║
╚═════════════════════════════════════════════════════════╝
```
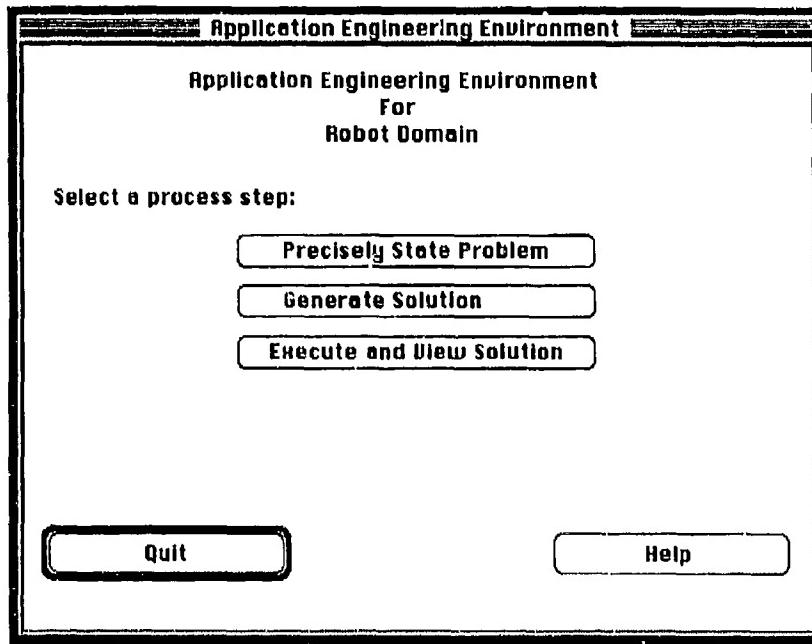
You will follow the application engineering process by selecting each of the three buttons, from top to bottom:

- The Precisely State Problem button assists you in precisely stating the customer's problem—that is, making decisions and validating the problem statement.

- The Generate Solution button generates a solution based on your statement of the problem.

- The Execute and View Solution button allows you to simulate execution of a robot, using a modified version of the Karel-the-Robot simulator program (please note that the programs you generate will not work with any Karel-the-Robot simulator you have used previously). You may also use the bottom button to view the software (solution) you generate.

If you need help, click on the Help button. When you are finished, click on the Quit button.

### 1. PRECISELY STATE PROBLEM

Clicking on the Precisely State Problem button in the main dialog box gives you the following dialog:

```
╔══════════════════════════════════════════════════════════════╗
║  ═══════════════════ Precisely State Problem ═══════════════   ║
╟──────────────────────────────────────────────────────────────╢
║                                                                ║
║                    Precisely State Problem                     ║
║                                                                ║
║                                                                ║
║   Select a process step:                                       ║
║                                                                ║
║                    ┌──────────────────────────┐               ║
║                    │     Make Decisions        │               ║
║                    ├──────────────────────────┤               ║
║                    │     Validate Decisions    │               ║
║                    └──────────────────────────┘               ║
║                                                                ║
║                                                                ║
║       Ready to make decisions.                                 ║
║                                                                ║
║                                                                ║
║   ┌──────────────┐     ┌──────────────┐     ┌──────────────┐  ║
║   │    Done      │     │    Cancel    │     │    Help      │  ║
║   └──────────────┘     └──────────────┘     └──────────────┘  ║
║                                                                ║
╚══════════════════════════════════════════════════════════════╝
```
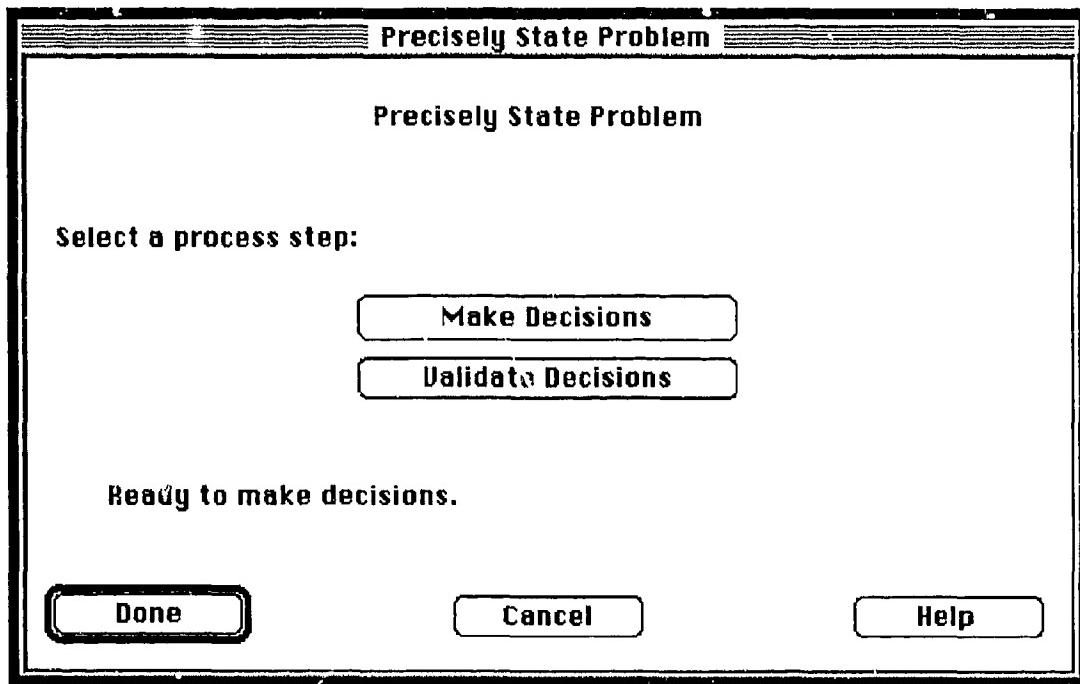
Use this dialog to make the decisions needed to identify a robot that meets a customer's needs (the Make Decisions button) and to validate the decisions you have made (the Validate Decisions button). Once you have performed these two steps, click on the Done button; you will return to the main dialog box. If, after performing either step, you feel you have made the wrong set of decisions, click on the Cancel button. You will return to the main dialog box, but you must make the decisions again (unless you had previously made and validated decisions, in which case those decisions are still in effect).

## Make Decisions

Clicking on the Make Decisions button in the Precisely State Problem dialog box lets you step through the decision-making portion of the application engineering process shown in class:
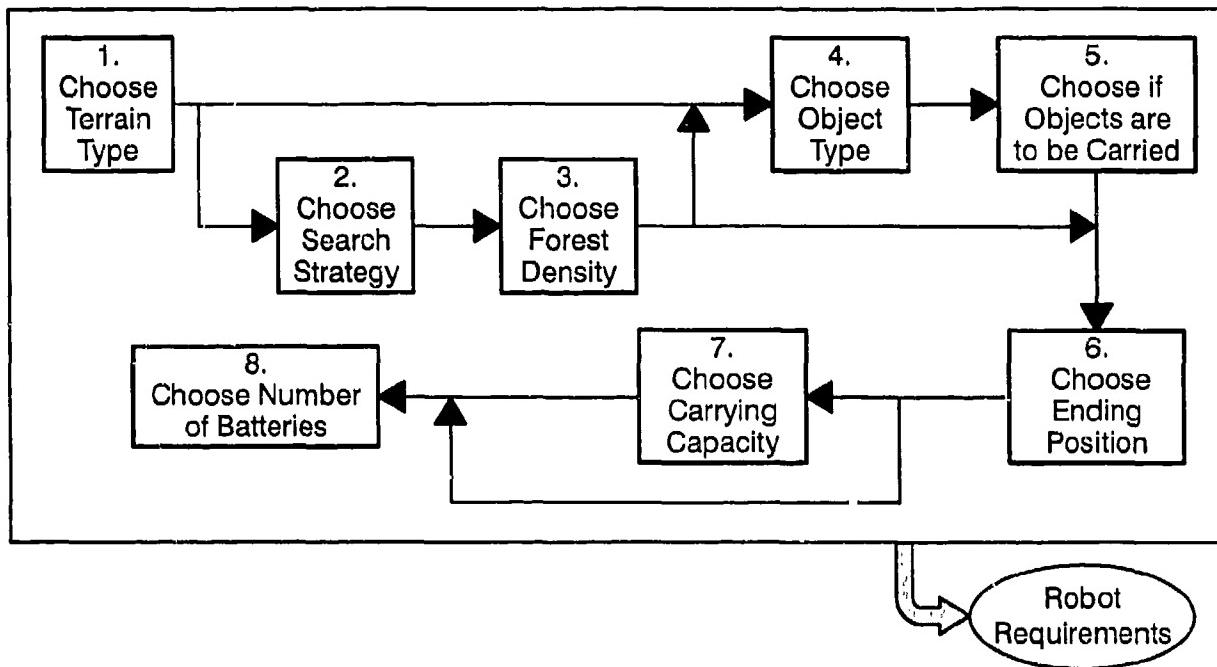
You will be presented with a dialog that lets you make each decision. You must make them in the order shown in this picture. You must first choose the terrain. You must then choose the search strategy, then the forest density, then the object type, and so forth. By making the choices in this order, you are following a logical process that is intuitive in the domain.

Sometimes, your choice determines which decision you must make next. Consider the search strategy. In some terrains, URW's engineers can determine the best search strategy. In those terrains, you will not be asked to choose the search strategy; the application engineering environment will choose it for you. Similarly, you only need to choose the forest density if you have selected forest as the terrain. The environment will guide you through the process, telling you the next decision to make.

Clicking on the Make Decisions button gives you the following dialog box:

```
╔═════════════════════ Make Decisions ═══════════════════════╗
║                                                            ║
║                       Make Decisions                        ║
║                                                            ║
║    ┌──────────────────┐                                    ║
║    │      Terrain     │                                    ║
║    └──────────────────┘                                    ║
║    ┌──────────────────┐                                    ║
║    │  Search Strategy │                                    ║
║    └──────────────────┘                                    ║
║    ┌──────────────────┐                                    ║
║    │  Forest Density  │                                    ║
║    └──────────────────┘                                    ║
║    ┌──────────────────┐                                    ║
║    │   Object Type    │                                    ║
║    └──────────────────┘                                    ║
║    ┌──────────────────┐                                    ║
║    │  Carry Objects   │                                    ║
║    └──────────────────┘                                    ║
║    ┌──────────────────┐                                    ║
║    │ Ending Position  │                                    ║
║    └──────────────────┘                                    ║
║    ┌──────────────────┐                                    ║
║    │ Carrying Capacity│                                    ║
║    └──────────────────┘                                    ║
║    ┌──────────────────┐                                    ║
║    │    Batteries     │                                    ║
║    └──────────────────┘                                    ║
║                                                            ║
║  ┌──────────┐      ┌──────────┐         ┌──────────┐       ║
║  │   Done   │      │  Cancel  │         │   Help   │       ║
║  └──────────┘      └──────────┘         └──────────┘       ║
╚════════════════════════════════════════════════════════════╝
```
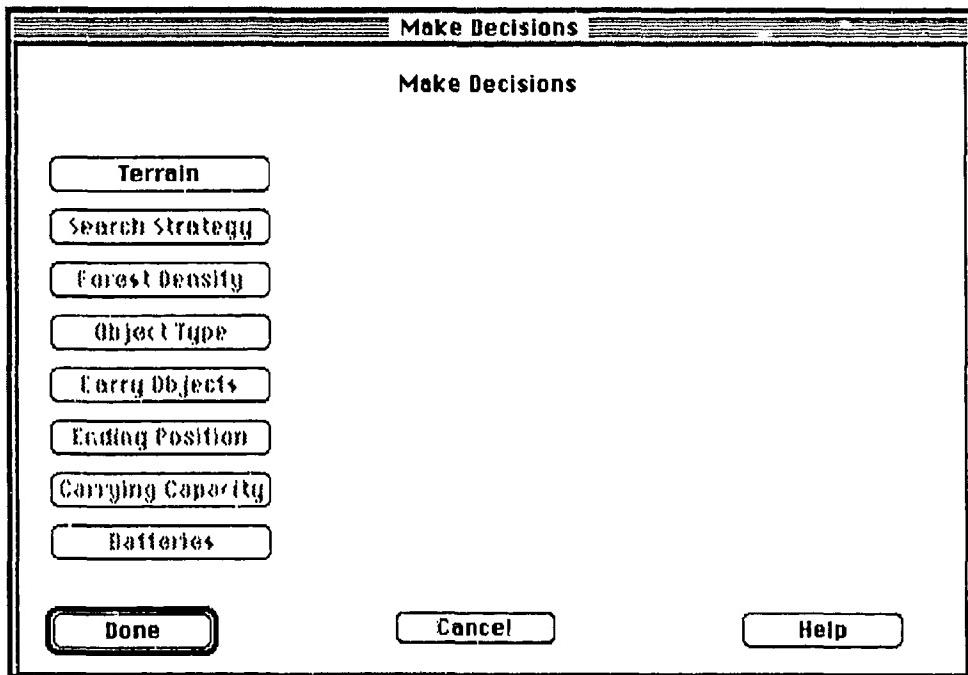
This dialog shows all the decisions you need to make. For each decision, it shows a button whose label is the decision's name. If you have made the decision, it shows the decision's value to the right of its button. Here, none of the decisions have been made, so no values are shown.

You will make the decisions in the order shown in the dialog box. You begin by clicking on Terrain. In most cases, you will be shown a dialog box containing a set of radio buttons that give the valid alternatives for the decision. For example, if you click on the Terrain button, the following dialog box will appear on your screen:

```
╔═══════════ Decision 1 - Terrain Type ═══════════╗
║                                                  ║
║  In what type of terrain will the robot operate?   ┌─ OK ─┐   ║
║                                                    └──────┘   ║
║     ○ Field                                        ┌ Cancel ┐ ║
║     ○ Forest                                       └────────┘ ║
║     ○ Tundra                                       ┌  Help  ┐ ║
║                                                    └────────┘ ║
║  Implications:                                   ║
║                                                  ║
║                                                  ║
║                                                  ║
║                                                  ║
║                                                  ║
║                                                  ║
║                                                  ║
╚══════════════════════════════════════════════════╝
```
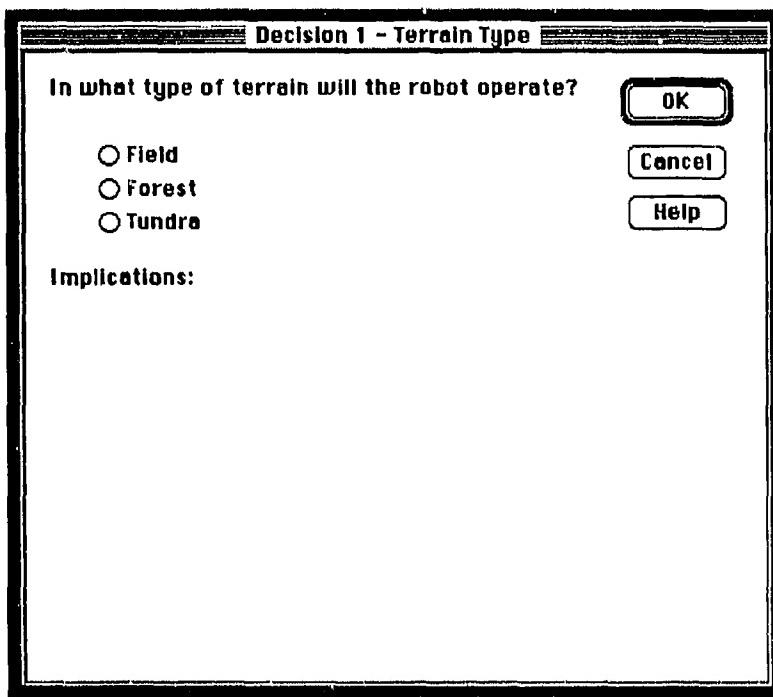
Click on the radio button labeled with the terrain your customer needs. Then click on the OK button, and you will return to the Make Decisions dialog box. Continue clicking on the buttons until you have made all the decisions.

For the Carrying Capacity and Batteries decisions, you will be asked to enter an integer value in a dialog box. Enter a valid integer, then click on the OK button.

Once you have begun making decisions, you must make all the decisions (that is, finish the entire process) before you can quit. When you have made all the decisions and think they meet your customer's requirements, click on the Done button. You will return to the Precisely State Problem dialog box.

You can abort the decision-making process by clicking on the Cancel button. If you abort the decision-making process, you must make the all decisions again.

Each time you make a decision, you will be shown the implications of that decision. These implications are presented in terms of how they affect the robot's hardware and software. You can feel free to experiment with different combinations of decisions. You should be able to see how different customer needs result in different robots.

When you make the Batteries decision, you will probably need help estimating how many batteries your robot will require. Click on the Help button, and you will be shown some values. Bear in mind that these are estimates. Depending on the terrain in which your robot operates—specifically, the distribution of objects and obstacles—your robot may actually need more or less energy for a particular mission. Keep in mind the consequences of failure as you choose the number of batteries. A robot that runs out of energy before picking up all litter is a nuisance. A robot that runs out of energy before reaching a stranded party of hikers can have tragic consequences.

## Validate Decisions

Once you have made all decisions, you are ready to validate your problem statement. In fact, much of the validation is already done. During decision making, you could not state a carrying capacity for a robot that does not pick up objects—the application engineering environment will not allow it.

However, you might still have made mistakes. For example, you could have misunderstood your customer's requirements and how they relate to the decisions. During validation, you are asked to review the decisions you have made. This is the time when you should make sure they are proper. To validate your decisions, click on the Validate Decisions button in the Precisely State Problem dialog box. A dialog box with the following information will appear on your screen:

```
                          Validation Report

For the following decisions, you should be able to see the values you chose in
the requirements your customer gave you:
        Terrain: forest
        Forest Density: average
        Object Type: litter
        Carry Objects: yes
Make sure these are what your customer wants!

The robot's cost is as follows:
        Cost of basic robot: $8888.00 (Including 1 battery)
            Carrying Capacity: $125.00 (250 pieces of litter @ $25.00 per 50 pieces of
                                litter)
        Number of Batteries: $2660.00 (20 batteries @ $133.00 per battery)
                Total cost: $11673.00
Here is some information on how your decisions influenced the cost.

                    CARRYING CAPACITY       NUMBER OF BATTERIES
                    Value   Price           Value   Price
        +10%        275     $11698.00        23     $11939.00
        Your decision   250     $11673.00    21     $11673.00
        -10%        225     $11673.00        18     $11274.00

                          [  Done  ]
```

During validation, you are also told how much the robot will cost. Unless your customer has a very deep wallet, you should check to make sure that the robot's price is within the customer's range.

If you decide that your decisions are improper, you can easily revise them. Exit validation (click on the Dismiss Report button), and click on the Make Decisions button again. This time, you will see the decisions you made previously rather than a set of decisions waiting to be made. You can change any decision; you do not need to start with the Terrain decision and make all the decisions again. However, if you change a decision, you must remake all the decisions that follow it in the application engineering process.

When you think you have a valid set of decisions, click on the Done button when you see the Precisely State Problem dialog displayed. This will return you to the main dialog box.

## 2. GENERATE SOLUTION

Once you are satisfied with your statement of the problem, you can generate a solution to it. Just click on the Generate Solution button in the main dialog box. The application engineering environment will choose all the correct parts for your solution and assemble them into a working program. When generation begins, you will see a calling hierarchy generalized for the domain. Each box represents a Karel-the-Robot instruction. Each line from instruction A to instruction B means that A invokes B.

You will see this hierarchy adapted to your decisions. (For example, if your ending position does not include returning, the portions related to returning will be deleted.) You will then see each instruction adapted. You will notice your cursor changing; this means the adaptation is happening. You will also see that the boxes start with black outlines, but end with gray. This shows the instruction has been adapted. The text inside each box suggests the nature of the adaptation.

When this step is finished, you are ready to simulate the execution of your robot.

## 3. EXECUTE AND VIEW SOLUTION

You are now ready to compile and execute the software you have just generated. Click on the Execute and View Solution button in the main dialog box. This invokes the modified Karel simulator

When you invoke the Karel simulator, you will see the following screen:
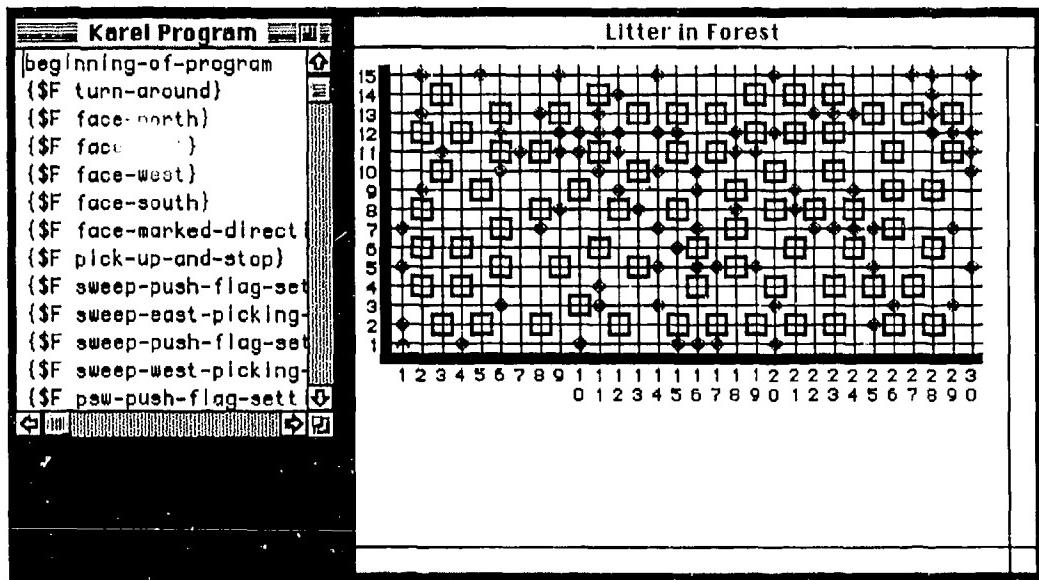


The Karel simulator has two windows: the Karel Program window and the World window. The Karel Program window holds a Karel program, such as the one you have generated. The World window displays a grid representing the robot and the terrain through which it traverses. You will see different titles for the World window depending on the terrain in which your robot is operating. Here, it is titled "Litter in Forest," meaning that the robot is operating in a forest scattered with litter. The robot is shown as a " ⊃ " symbol, oriented in the direction the robot is headed; here, it is at location (1,1) in the grid, and facing north. Objects are shown as stars aligned on grid intersections, such as at (6,3). Trees and rocks are shown as squares centered around grid intersections; the robot must negotiate its way around these.

You may want to examine the program you have generated. You can do this through the Karel Program window. It is a standard Macintosh text window. You can move, enlarge, shrink, or scroll through it using the mouse. However, you cannot modify its contents.

## 3.1 COMPILE YOUR PROGRAM

You must compile your program before you can execute it. Select Compile from the Run menu. The Karel simulator will compile your program. It should compile without any errors. (You may also skip this step and choose Run. If you do, the simulator will automatically compile your program before running it.)

## 3.2 RUN THE COMPILED PROGRAM

You can now execute your program. Choose Run from the Run menu. You will see a message telling you the world is being restored. The robot will then begin executing its instructions.

As the robot runs, the simulator displays the robot's remaining energy and the number of objects the robot is currently carrying (if it carries objects) in the lower right of the World window. When the robot stops running, the simulator displays the elapsed execution time. You can use this data to complete the laboratory.

You will usually want to run the robot in this manner. However, you can control certain properties of the robot's execution. You can:

- Control the speed at which the robot executes each instruction

- Pause the robot as it executes

- Cause the simulator to show, in the Karel Program window, the next instruction to be executed

- Make the robot execute a single instruction, then pause to wait for your next command

To do any of these, you must first select the World window. Position the cursor anywhere in it and click the mouse button. Your screen will look similar to the following:

```
     Karel Program                    Litter in Forest
beginning-of-program
($F turn-around)
($F face-north)
($F face-east)
($F face-west)
($F face-south)
($F face-marked-direct
($F pick-up-and-stop)
($F sweep-push-flag-set
($F sweep-east-picking-
($F sweep-push-flag-set
($F sweep-west-picking-
($F psw-push-flag-sett

     Stop        Pause        Energy Remaining: 14364 KJ•
                              Objects In Bag: 12
                              Execution Speed
```
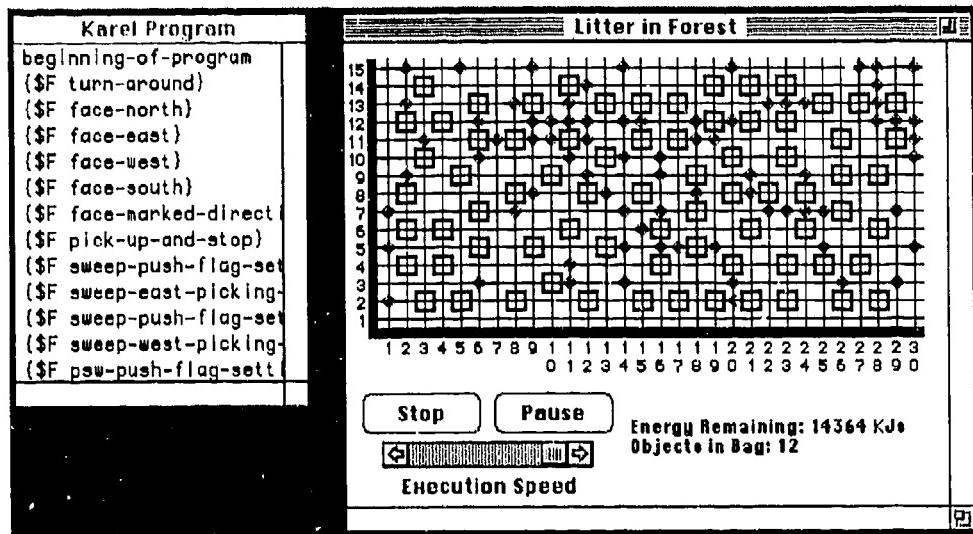
Here, the robot is at location (20,2) on the grid, and facing west.

## Changing the Execution Speed

The World window has an Execution Speed control. By default, your robot moves at its top speed. You can slow down the robot by moving the Execution Speed control to the left. (When you are doing the laboratory exercises that ask you to compare execution times in different solutions to the same problem, be sure the Execution Speed control is in the same position each time you run the robot.)

## Pausing the Robot

You may temporarily stop the robot at any time by clicking on the Pause button. When you do, the robot will stop, and Pause will change to Continue. Click on the Continue button to continue execution.

## Showing the Next Instruction to Execute

If you wish, you may observe the thread of control as the robot executes. Select Trace Execution from the Options menu. The program will highlight, in the Karel Program window, each instruction the robot executes. You may turn this option on and off any time during execution of the robot.

## Executing a Single Instruction

You can command the program to execute a single instruction, then pause and await your orders. You do this by selecting Single Step from the Run menu. Each time you chose Single Step, the robot will execute a single instruction. After it executes the instruction, you may either continue executing by single steps, or choose Run and have the robot not pause after each instruction.

# UNIT 3: APPLICATION ENGINEERING LABORATORY

## TEACHER NOTES FOR LABORATORY

*Comment:* This laboratory lets students use an application engineering environment. The environment implements the application engineering process for the robot domain covered in the Unit 3 lecture.

The environment plays down the role of programming. Students create programs, but not as they have previously. Instead, the environment automatically creates the software based on a problem statement the student provides. Theoretically, students can perform this laboratory without ever seeing any software. The environment contains a tool that lets them do so; this emphasizes that software is necessary to the robot but that it can be developed in more than one way.

What substitutes for programming is:

- Eliciting and understanding customer requirements and elaborating them in terms of the domain problem space. The result is a precise problem statement.

- Quantitative and qualitative analysis of requirements to determine satisfaction of customer needs.

- Simulation as a means to validate customer requirements.

The second item is most significant and probably less intuitive to students than the others. Students are asked to study problems and certain properties of solutions purely in terms of domain problem space concepts. They are not allowed to think in terms of primitive Karel instructions or even algorithms. They must act as application engineers, not programmers. By having them do so, you can demonstrate to them that programming is only a means to an end, not an end in itself.

## 1. CORNFIELD ROBOT

*Answer to Question A:* This question asks the student to analyze a problem without first trying to generate a solution to that problem. The application engineering environment presents all the information the student needs. The student must first precisely state the problem, selecting "field" as the terrain; this fixes the decisions on search strategy and object type and obviates the decision on forest density. If any students wonder why, you can explain it to them as decisions already made by the domain engineers:

- In fields, URW only knows how to build robots that harvest corn. It doesn't possess the technology to build robots that mechanically harvest, for example, tomatoes.

- The domain engineers' studies have concluded that sweeping is a more efficient strategy than zigzagging when harvesting corn. (Real harvesting machines work this way.)

The student must choose the most appropriate robot. The assignment defines this as the robot that costs least, but can still perform its mission. Since carrying capacity and number of batteries are the

two factors that determine a robot's cost, the student must experiment with variations of these quantities to complete the assignment. The students will simply have to try several values of carrying capacity. They can determine the number of batteries through the help facility (available by clicking on the Help button). During validation, they can obtain the cost of the robot they have modeled. Using this information, they should create a graph similar to the following:
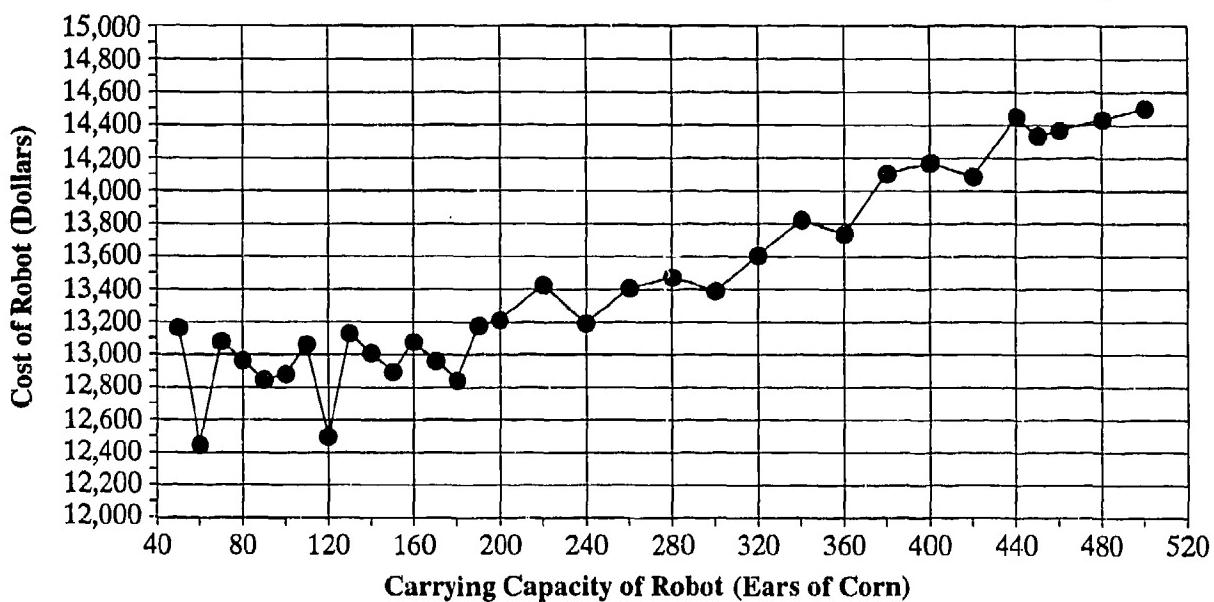


Inform the students that they will need some strategy for choosing carrying capacities, unless they are so motivated as to try all 451 possible values. Note the trend of cost increasing as a function of carrying capacity. This should motivate them to try a binary search strategy. Binary search by itself is not adequate, because the robot's cost does not increase monotonically as a function of carrying capacity, but it is a good start.

For this mission, the robot costs least when its carrying capacity is 60. Here is the reason why. In the cornfield, each location contains one ear of corn. Therefore, each row has 30 ears. Any multiple of 60 minimizes the number of spaces a robot must move to unload its cargo and return to continue harvesting, since it always fills its bag when it is against the western border. A value that is not a multiple of 60 would require the robot to move west as well as south as it returns. Each move consumes battery power, necessitating extra energy; since multiples of 60 minimize moves, they are preferred. Note that the robot will make fewer moves if its carrying capacity is 120 instead of 60, and indeed will make the fewest moves if its carrying capacity is 449 (the number of ears of corn in the field). However, extra carrying capacity costs money, and carrying 400-plus ears increases the robot's weight enough to cause it to consume energy rapidly. This in turn requires extra batteries, driving up the robot's price. For these reasons, 60 is the optimal carrying capacity.

This fact—that robots in cornfields behave best when their carrying capacity is a multiple of 60—is an excellent example of the type of knowledge possessed by experts in a domain. That is, it is something an application engineer would know and would automatically apply when approached by a customer. This knowledge would be gained by experience, through trial and error. Deriving it mathematically is difficult; in many domains, it is impossible. Eventually, application engineers feed this type of

trial-and-error experience back into domain engineering, where experts incorporate it as a heuristic in the application engineering environment.

You can discuss this with students. Ask them for commonplace but significant knowledge in other domains. A few examples: does your automobile owner's manual tell you how to park your car? Few do; of those that do, do any tell you to put money in the parking meter? Does your owner's manual say to turn off your ignition after you park your car?

*Answer to Question B:* The following are some sample results:

| Carrying Capacity (Ears of Corn) | Execution Time (Seconds) |
| --- | --- |
| 50 | 1:00.25 |
| 60 | 46.14 |
| 500 | 33.84 |

These numbers were obtained running the Karel simulator on a 486-based computer. The numbers you obtain will depend upon the computer you use. However, you should still obtain the same ordering: a carrying capacity of 50 results in the slowest execution time, and a capacity of 500 results in the fastest. Therefore, if your customer wants a robot that can harvest corn as quickly as possible, and if money is no object to her, you should recommend that she choose the robot with the greatest carrying capacity. The most alert student will also observe that, as a field contains at most 449 ears of corn, the customer could save a little money without sacrificing execution speed by buying a robot whose carrying capacity is 449 or 450 (both these robots cost the same).

You are not actually running a robot; you are running a simulation. If URW were a real company, the application engineer would run a simulation such as this to learn facts about the robot's performance that cannot be determined in other ways (i.e., as part of validation). This point is well-illustrated in laboratory Questions 2 and 3, with their somewhat randomly-placed objects and obstacles. Addressing the issues raised by Questions 2 and 3 by deriving formulas is very hard. Simulation provides a simpler alternative.

## 2. RESCUING ROBOT

*Answer:* There is no point in deciding that a robot should pick up hikers and continue until it runs out of energy. The purpose of a "rescue" mission would be either to transport the hikers to a safe, known place (either the origin or the point where the entire terrain has been covered—both can be predicted) or to stay with the hikers until help arrives. If the robot continued until it ran out of energy, the National Guard would have difficulty locating it, so the hikers would be no better off than if they had just stayed where they were.

This laboratory comes with two maps of a tundra. One, named Tundra #1, is intended to illustrate the average case. A group of three hikers is stranded more or less in the middle. The other map, named Tundra #2, illustrates the worst case. There are a total of five hikers (the maximum permissible carrying capacity). Four are right at the beginning of the robot's search. The remaining hiker is at the very end. Suppose you opt to have the robot pick up hikers and return. The robot will consume the maximum possible amount of energy. It must carry four hikers the greatest possible distance before it completes its search by finding the fifth. Since carrying an object consumes energy, the robot's energy use is maximized.

Choosing instead to have the robot stop when it locates a person creates a robot that is probably unsatisfactory. It will find one group of hikers but not the other. This is not likely to please the National Guard, nor is a robot with a carrying capacity so small that it returns before it finds everyone.

The purpose of this question, then, is to make sure the students study the problem carefully and truly understand the needs of their customer. They must pay particular attention to the following:

- Only certain combinations of ending location and carrying capability are useful for rescuing people.

- The robot must not run out of energy. In a cornfield, the consequences of doing so are annoying. In a tundra, human lives are at stake. Failure has dire consequences.

- Application engineers must make important choices based on their own judgement. The application engineering environment cannot calculate the right amount of energy. It can predict average use (note that the robot will actually fail if given the average number of batteries needed: the hikers are just a bit beyond the midpoint, which is assumed to be average), and it can predict worst-case use. The worst-case robot works but is very expensive. Most customers are not willing to pay the price on the off-chance that the worst case will occur. They want something that handles most cases. The application engineer has the moral responsibility to present this information to the customer and to try to come up with the best energy statement. In the laboratory, you might want to act as customer and establish an arbitrary price ceiling that precludes building the worst-case robot. As part of the assignment, ask the students to prepare a report of what they expect the robot can do.

## 3. LITTER-GATHERING ROBOT

*Answer:* The following table was created using a carrying capacity of 250, with the robot picking up litter and returning to its point of origin when its bag is full. The average-case number of batteries was used. Such a robot does not have enough energy to complete is mission, but the general trend illustrated by the table does not change with the number of batteries.

| Density of Trees in Forest | Cost of Robot | |
|---|---|---|
| | Search by Sweeping | Search by Zigzagging |
| sparse | $9,944.00 | $11,806.00 |
| average | $10,368.00 | $11,675.00 |
| dense | $10,875.00 | $11,540.00 |

Notice the difference between the columns. The cost of a robot that sweeps is proportional to the forest density. The cost of a robot that zigzags is inversely proportional to forest density. If you examine the code you will observe that navigating around a tree in a sweep requires two extra moves and eight extra turns. By contrast, zigzagging around a tree requires four fewer turns than if the tree were not present. In theory, then, a robot moving in an extremely dense forest (or a larger one) would do better to zigzag. In practice, a Karel map cannot contain enough trees to make this worthwhile.

## START OF FOURTH UNIT

## DISCUSSION

This unit describes the concepts underlying the tools used in the laboratory exercise.

Domain engineers develop a "black box" (in fact, you used a black box in the laboratory exercise); the application engineer uses the black box when creating a solution for a customer. The application engineer doesn't need to know what is inside the black box to use it.

Unit 4 opens up some of the black box. It gives a brief introduction to the major products created during domain engineering:

- The application engineering process

- The description of what problems are in a domain

- The domain architecture

- The architectural components

The unit cannot open up all of the black box. It deliberately avoids issues of representation, because of time constraints. It does not attempt to cover how to do domain engineering; it discusses only the results and how they relate to application engineering.

This unit relates megaprogramming to the skills that the students have already acquired in class (that is, domain engineering requires programming skills).

## OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Understand some of the concepts underlying the tools used in the Unit 3 laboratory

- Relate megaprogramming to the programming skills they have acquired

# Unit 4: Domain Engineering

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Transfer*

4-1

## DISCUSSION

This slide shows that application engineers produce robots through a process supported by a **black box** called **process support**. The laboratory from Unit 3 required students to use this black box.

Therefore, we can think of application engineering as requiring:

- A process (the steps for producing robots and the robot software)

- Process support:

  - Karel instructions (software procedures and functions)

  - Automated tools that support the process (like the programs in the laboratory that helped you state problems and generate solutions)

The laboratory showed how use of these made producing robots straightforward.

Process and process support are created during domain engineering. How did they come to be?

- How did domain engineers decide what the process to precisely state the problem should be?

- How did the domain engineers decide what process support to build?

- For that matter, how did domain engineers decide what the domain should be? Why did they include some robots and not others? For instance, why were only three terrains allowed?

Let's look at each of these questions, starting with the last one.

## STUDENT INTERACTIONS

- Do you think process and process support exist in other domains? (A **very** good example is a fast-food restaurant.)

## OBJECTIVE

The students should be able to:

- Explain the application engineering process

# What Help Do Application Engineers Need?

Application Engineer

Step 1: Precisely State Problem

Robot Requirements

Step 2: Generate Solution

object sensor

compass

rock-avoiding algorithm

zig-zag

Process Support
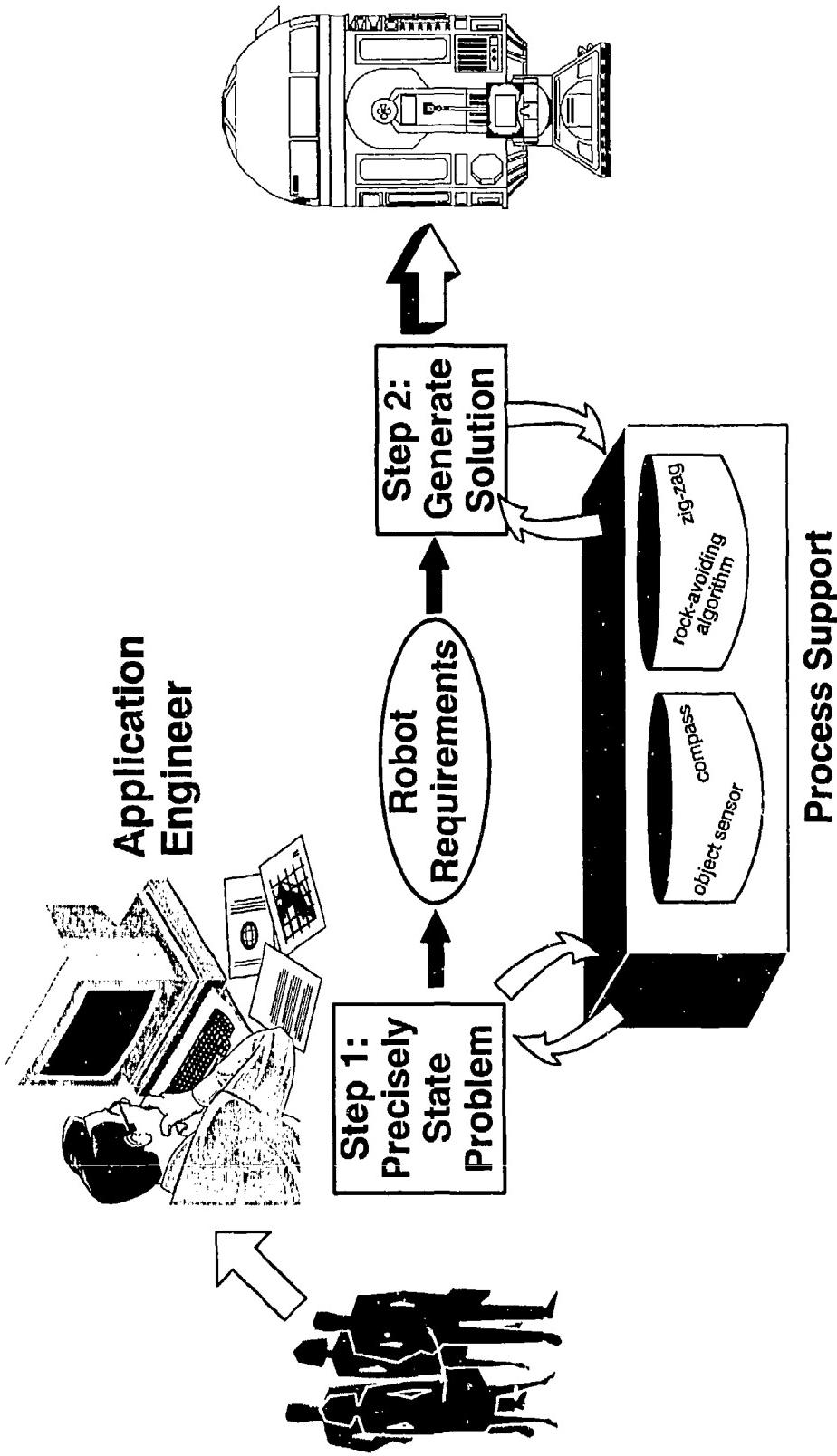
## DISCUSSION

This slide presents examples of factors (in terms of our robot domain) that domain engineers consider when they decide what problems and solutions to include in a domain.

It's domain engineers, not application engineers, who decide what's in a domain. Application engineers create individual systems. Domain engineers are responsible for deciding the range of systems that application engineers can create.

In our robot company, the domain engineers might think along the following lines:

- We know how to build robots that operate autonomously, and we think our customers would be happier if they didn't have to control a robot as it operates.

- We think that the demand for robots that search for objects is larger than the demand for robots that place objects (an example of the latter is a robot that sows seeds).

- We've built robots that operate in a tundra, a forest, and a field. We can continue to do so without investing lots of money.

These are the types of factors that constrain the problems and solutions that form the domain (notice there are both business and technical factors). Part of domain engineering is studying these factors and deciding what problems and solutions are important for a successful business.

Once the domain engineers know the problems that will be in the domain, they can study them and uncover the commonalities among all problems and the differences between instances of problems.

This is the basis for the decision-making process we saw in the Unit 3 laboratory. What sort of information helps domain engineers create this process?

## STUDENT INTERACTIONS

- How do General Motors, Ford, and Chrysler decide what types of cars to build (that is, what kind of cars will be in their domain)? Are their domains completely the same or are there differences?

## OBJECTIVE

The students should be able to:

- Explain the types of information domain engineers use to determine which problems (and which solutions to those problems) should be in a domain and which should be excluded

# Factors Influencing
## What Is In a Robot Domain

- The knowledge about robots you possess

- The types of robots your company has built

- The types of robots you think your customers want

- The types of robots your boss wants you to build

# DISCUSSION

Domain engineers create a decision-making process based on what's common and what's variable in the domain. They deliberately exclude from this process all the things that are common—if something is common to all problems and solutions, there's no decision to be made. For example, you weren't asked in the lab if your robot needed a locomotion mechanism because all robots need a locomotion mechanism.

This slide, then, shows the type of information that helps domain engineers to reason, systematically, about a process for describing robots. They will later use this information to create a process that the application engineer can follow.

This is the type of information that you would know if you were familiar with the domain. (In fact, most of it makes good sense even if you're seeing the domain for the first time.) Domain engineers are familiar with the domain. They would have no trouble listing facts such as these.

The list is not ordered. Domain engineers create such lists using stream-of-consciousness thinking. They must subsequently organize it, but at first they simply try to enumerate their knowledge of the domain.

**Remember again: this information emphasizes what varies between robots. Making decisions is (by definition) based on variations, not on commonalities.**

How do we use information of this sort to help us create a process?

# STUDENT INTERACTIONS

* What is some of the information that could be used to create a decision-making process for choosing an automobile (e.g., color, four-wheel drive, number of doors, engine size, etc.)?

# OBJECTIVES

The students should be able to:

* Explain the type of knowledge that goes into creating a decision-making process

* Explain that making decisions is based on variations, not commonalities

# Information That Helps You Create a (Robot) Decision-Making Process

- Robots whose mission is to locate objects but not carry them do not need arms or bags.

- Some objects cannot be carried; they might be too big or immovable.

- A terrain contains certain types of objects; no terrain contains all types of objects.

- Some terrains are much larger than others; we would not expect a cornfield the size of a tundra!

- . . .

Overview of Megaprogramming Course: Unit 4, Domain Engineering. Lecture

*Virginia*
**CENTER of**
**EXCELLENCE**
*for Software Reuse and Technology Transfer*

4-4

## DISCUSSION

This slide covers issues in creating a process, emphasizing the decision-making portion of the process.

At a minimum, the process must ensure that application engineers make all relevant decisions, validate the decisions, and generate the software. Domain engineers use the information about robots (see previous slide) to formulate the decision-making process. In any domain, some decisions influence others. Here are two examples in the robot domain:

- Whether objects should be carried (Decision 5) influences carrying capacity (Decision 7).
- The terrain (Decision 1) influences the object type (Decision 4) and number of batteries (Decision 8).

The decision-making process is ordered to reflect these influences. (This is not the only valid ordering.) You are used to validating your programs after you generate them. In our robot domain, you do it before! We are able to validate before generation, because we know that the decisions are an accurate model of the software. The earlier we validate, the less effort we waste if we make mistakes.

In the laboratory, most of the process was automated. A program asked you to make decisions in a particular order, helped you validate your requirements, and generated the software for you. Domain engineers should try to automate as much as they can. The application engineers provide feedback to the domain engineers on things that did and did not work with the process support. The domain engineers (both domain and application) working to improve the process support based on these comments. With many engineers (both domain and application) working to improve the process support, errors are more likely to be found quickly.

Through domain engineering, the domain engineer has made all these things pretty much "mechanical" and unambiguous. The domain engineer tries to provide **process support** that makes the computer do whatever is mechanical. Anything that requires creativity, the application engineer must solve independently.

Now, how do we go from the decision-making process to generating a solution?

## STUDENT INTERACTIONS

- Is the decision-making process mechanical for choosing an automobile? Could it be?

## OBJECTIVES

The students should be able to:

- Explain the type of knowledge needed to create a process
- Understand the importance of automation in process support

# Defining a Process for Precisely Stating a Problem



**Step 1: Precisely State Problem**

1. Choose Terrain Type
2. Choose Search Strategy
3. Choose Forest Density
4. Choose Object Type
5. Choose if Objects are to be Carried
6. Choose Ending Position
7. Choose Carrying Capacity
8. Choose Number of Batteries

*Determined by variations among robots*

Robot Requirements

Validate Requirements

*Validate as early as possible*

*Let the application engineer correct mistakes*

**Step 2: Generate Solution**

*Virginia*
**CENTER of EXCELLENCE**
*for Software Reuse and Technology Transfer*

## DISCUSSION

This slide begins a discussion (ending with 4-11) of how domain engineers define Step 2 of the Application Engineering process (generating a solution).

Domain engineers do three things to allow application engineers to generate a solution. First, they must develop an understanding of the structure of all programs for robots in the domain: what structural characteristics all programs have in common and how the characteristics vary. We call such a structure a **software architecture**. This slide presents, as an architecture, a calling hierarchy, showing the basic structure of robot software and suggesting some variation (e.g., forests and tundras have obstacles; fields don't).

Second, the domain engineers must develop **reusable code components** that can be used in this architecture. The architecture defines the Karel instructions needed, and their interrelationship, for a particular program. The domain engineer creates those instructions so the application engineers have them available when they want to create a robot.

Third, the domain engineers must define **generation procedures**. These describe how, for a particular problem, the architecture and the code components must be adapted to provide a solution to that problem. (Components as well as architecture must be adapted; for instance, Handle Object will need to be adapted to the type of object for which the robot is searching.) The generation procedures also describe how to fit the components into the architecture to form a complete program.

## STUDENT INTERACTIONS

- Where do the reusable code components come from? Could they be obtained from existing robot software? Would this be of any benefit?

## OBJECTIVES

The students should be able to:

- State what the domain engineer must do to define how application engineers generate a solution to a problem in the domain

- Explain that every program has an architecture

- Explain that an architecture shows a program structure

# Defining How the Application Engineer Generates a Solution

## The domain engineer must develop:

### Robot Software Architecture

- Perform Mission
- Return to Origin
- Navigate Terrain
- Negotiate Obstacle
- Handle Object

### Reusable Code Components

- Navigate Terrain
- Handle Objects
- Perform Mission
- Negotiate Obstacle
- Return to Origin
- . . .

### Generation Procedures
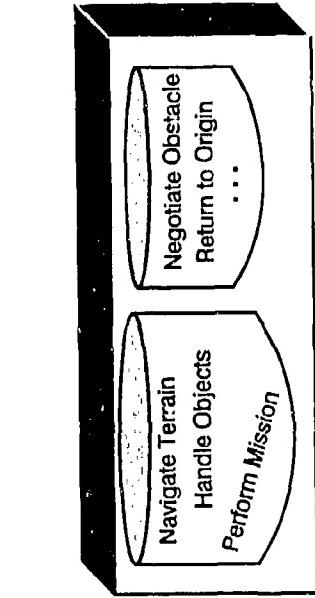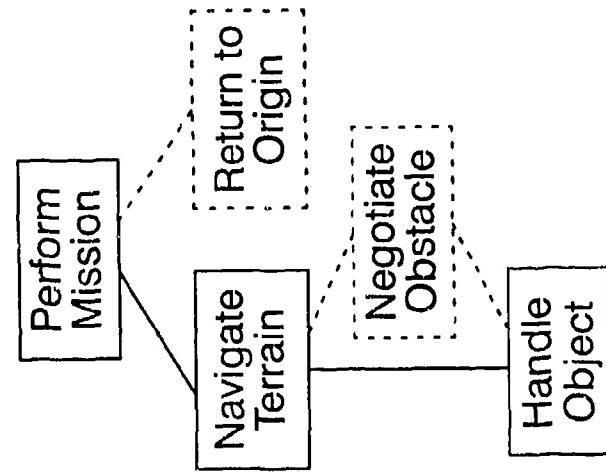
- Always need to perform mission
- Always need to navigate terrain
- If terrain is forest, need tree negotiation
- If terrain is tundra, need rock negotiation

## DISCUSSION

This slide begins a series that will show students how the domain engineer defines a software architecture for a domain. This slide begins by examining the architecture for a single program. Slides 4-8 and 4-9 are variations and generalizations of this one.

This slide is a detailed look at the software architecture of a particular robot, one which you were asked to generate in the Unit 3 laboratory (this should help you follow its logic). It shows the Karel instructions that make up the program (boxes), which instructions invoke which (lines), and the circumstances under which one instruction invokes another (commentary in italics). This slide shows only the major instructions (a slide with all instructions would be too crowded to be readable).

This architecture shows the general logic of the program. The main instruction is the one that drives the mission. This instruction does two things: it first invokes a zigzag toward the northeast corner of the forest and then invokes a return instruction. While zigzagging, it instructs the robot to pick up any litter. If it finds enough litter to fill the bag, it returns from that spot. This set of instructions is probably pretty similar to what you would write if your teacher asked you to create this one robot.

Now, how can we make use of this notion of architecture in building our robot domain? That is, how does an architecture help us generate a solution to any problem in the domain, not just to a single problem?

## STUDENT INTERACTIONS

- Did every program you write this year have a software architecture? In what part of the software development process should you develop the architecture (requirements, design, code, or test)?

- Does a box correspond to a single Karel instruction? In Pascal, are data types part of the boxes?

## OBJECTIVE

The students should be able to:

- Create a skeletal program based on an architecture diagram

# Robot Software Architecture for Robot in Forest That Picks up Litter

Mission: pick up litter and return

*(called when robot has returned to point of origin)* → Turn Off

*(called when all of forest is covered or when bag is full)* → Return to point of origin

*(called once robot reaches south border)* → Sweep toward west border, ignoring litter

*(called first)* → Sweep toward south border, ignoring litter

*(called to begin search for litter)* → Zig-zag toward northeast corner of a forest, looking for litter

*(called each time litter is found)* → Pick up a piece of litter

## DISCUSSION

This slide shows an architecture for a different robot than in Slide 4-7. Comparing and contrasting the two slides shows similarities and differences.

This robot searches a forest, but for hikers, not litter. It does not pick up hikers. Instead, it transmits their position when it locates them. Like the last robot, it returns to its point of origin when it reaches the northeast corner of the forest. Unlike the last robot, it does not turn off after returning. It sets out again, continuing this cycle until it runs out of energy.

This robot has an architecture similar to the previous one. For each box in one (except Turn Off), there is a corresponding box in the other. But note that on this slide:

- All boxes concerned with objects refer to people, not litter.

- The mission is different. This robot only searches for people. It does not pick them up. It searches in-definitely, not for a fixed period.

- A robot might transmit its position instead of picking up an object. This is a different reaction on locat-ing an object.

## STUDENT INTERACTIONS

- Is there any difference in the strategy a robot would use to search a forest if it's looking for people instead of litter?

- Does this robot's software for sweeping differ from the sweeping software in the robot on Slide 4-7?

## OBJECTIVE

The students should be able to:

- Explain the differences between this architecture and the architecture on the previous slide

# Robot Software Architecture for Robot Searching Forest for People



Mission: search for people in forest and return

*(called to begin search for people)*

Zig-zag toward northeast corner of a forest, looking for people

*(called each time a person is found)*

Transmit location

*(called when all of forest is covered)*

Return to point of origin

*(called once robot reaches south border)*

Sweep toward west border, ignoring people

*(called first)*

Sweep toward south border, ignoring people

## DISCUSSION

This slide shows how two architectures may be combined into one architecture that can be adapted into either of the two original architectures. It combines the architectures of the previous two slides.

Domain engineers need to understand the similarities and differences among architectures. This knowledge lets them understand which parts of the software can be reused in different robots.

This picture shows some of the similarities and differences. Abstract phrases describe components:

- Every robot performs a mission, but we've seen that this mission differs from one robot to another. Let's generalize the top box to "Perform Mission."

- Different robots search for different objects. Let's remove the references to the type of object for which the robot is searching.

Also, comparing Slides 4-7 and 4-8 shows that some components aren't always needed:

- Robots return under different circumstances. In fact, some don't return at all.

- Robots don't always turn off voluntarily. Some operate until they run out of energy.

In this slide, dashed boxes mean the component is used in only some architectures. Arrows with a solid line mean the instruction at the tail calls the instruction at the head in all programs in the domain. Arrows with a dashed line mean the call exists in only some programs. (If it doesn't exist, the instruction at the arrowhead isn't needed.)

What is a general architecture for any robot in the domain?

## STUDENT INTERACTIONS

- Can you see how to get from this generalized architecture back to the specific architecture? Be as precise as possible.

## OBJECTIVE

The students should be able to:

- Understand the transition from the specific robot architectures on the previous slides to the more general architecture depicted on this slide

# Combining the Two Software Architectures



```
                         (called if robot is to turn off)
                                            ┌ ─ ─ ─ ┐
                                              Turn Off
                                            └ ─ ─ ─ ┘

  ┌──────────┐                                              ┌──────────┐
  │ Perform  │ ──(called when mission is achieved)───────→ │Return to │
  │ Mission  │                                              │ Origin   │
  └──────────┘                                              └──────────┘
```

(called if robot is to turn off)

(called when mission is achieved)

(called to begin mission)

(called when an object is found)

(called once robot reaches south border)

(called first)

Turn Off

Perform Mission

Return to Origin

Sweep toward west border, ignoring objects

Sweep toward south border, ignoring objects

Zig-zag toward northeast corner of a forest

Handle Object

## DISCUSSION

This slide completes the generalization of the architecture.

The architecture shown here is one that domain engineers agree can be used for any robot in the domain.

- All robot software is driven by an instruction to perform the mission. One instance of that instruction is "Pick up litter and return." Another instance might be to "Stop when an ear of corn is found."

- All robots need an instruction that tells them how to navigate through the terrain. In a tundra or field, the robot will sweep. In a forest, the robot will sweep or zigzag.

- Robots in fields don't have obstacles; therefore, they don't need any instructions to negotiate their way around them. Robots in other terrains must face rocks, trees, etc.

- Some robots return; others do not. Therefore, not all robots need an instruction to return.

Each box corresponds to a reusable software component. An application engineer builds robot software by selecting:

- Each component required by all robots in the domain (here, "Perform Mission" and "Navigate Terrain"), adapted to the nuances of the robot (e.g., "Pick Up Litter and Return" for "Perform Mission").

- The proper set of components that are needed to support the particular robot the application engineers want, adapted to the mission (e.g., "Negotiate Rock" for "Negotiate Obstacle").

The domain engineers have to figure out which reusable components are needed for which robots. That's not hard—it's just a matter of figuring out the right combinations. These are the generation procedures mentioned on Slide 4-6.

The domain engineers also have to create the reusable components. This is where all the programming knowledge you've been taught is required. As a domain engineer, you still have to create the Karel instructions. But you do so for a whole domain of robots, not for a single one!

## OBJECTIVE

The students should be able to:

- Understand the generalization from a set of similar architectures to one that applies to any robot in the domain

Over[...] of Megaprogramming Course:   Unit 4, Domain Engineering, Lecture

4-10

*Virginia*
**CENTER of**
**EXCELLENCE**

*for Software Reuse and Technology T[...]*

# Robot Software Architecture for Robot Domain



**Perform Mission**
- Turn Off
- Return to Point of Origin
  - Navigate Terrain
  - Navigate Terrain
- Navigate Terrain
  - Negotiate Obstacle
  - Handle Object
  - Navigate Terrain

Key:

▢ Always present

⬚ Sometimes present

→ X calls Y in all programs

⇢ X calls Y in some programs
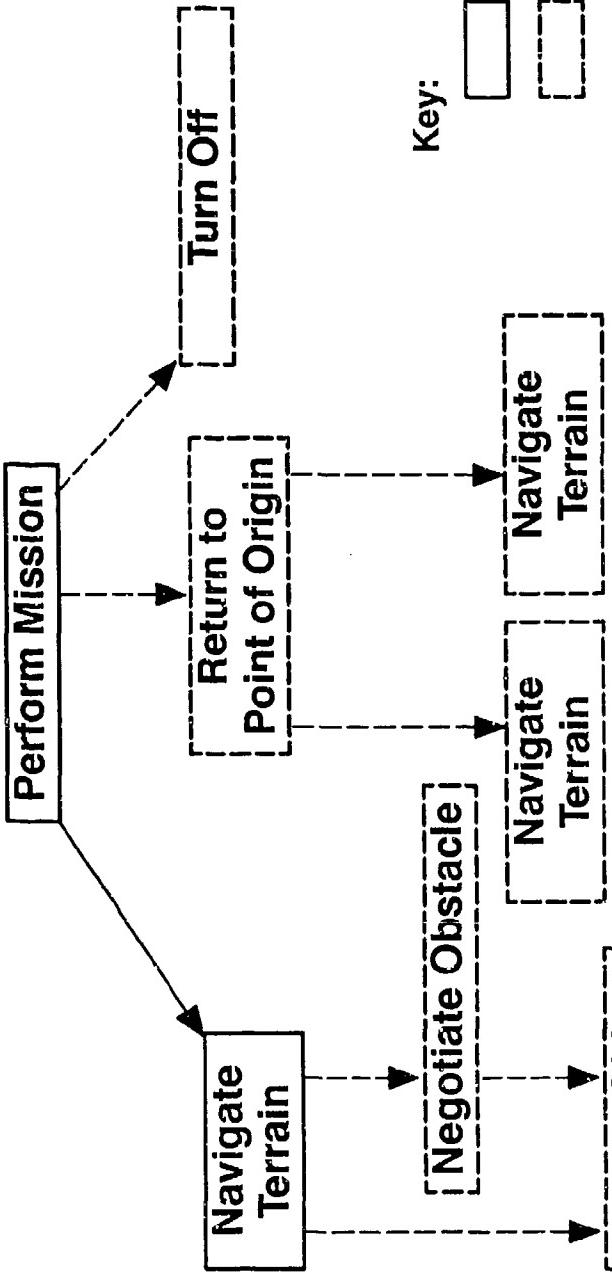
## DISCUSSION

This slide shows the solution generation capability that the domain engineer creates for the application engineer. The example is for a robot that picks up litter in a forest, returning when its bag is full or when it has covered the entire forest, whichever occurs first.

Assume the application engineer has made the set of decisions that correspond to this robot. This slide places in context the material contained in Unit 4:

- Domain engineers have figured out the architecture for the robot domain and have implemented a set of reusable components (shown in the buckets at the bottom left corner).

- They have also implemented a "component selector" box which, given a set of decisions (shown as the robot requirements), identifies the needed set of reusable components. These selected components are classes of instructions, rather than specific instructions. In this slide, we select a "Perform Mission" component, which is a component capable of supporting all possible missions a robot might perform. Because the robot zigzags, we don't select a component to negotiate an obstacle.

- Domain engineers have also implemented a "component adaptor" box which, given these decisions and the selected components, adapts the component to the decisions. In this slide, the "Perform Mission" component is adapted to perform the specific mission of picking up litter, returning when the bag is full. Similarly, the "Navigate Terrain" component is adapted to a "Zig-zag."

- The domain engineers have also implemented a "component composer" (the funnel), which takes all the adapted components and combines them into a single solution.

This is what happened when you selected "Generate Solution" in the application engineering environment.

## STUDENT INTERACTIONS

- Where does the robot software architecture (as shown on the previous slide) tie into this picture?

## OBJECTIVES

The students should be able to:

- Explain how domain engineers use the knowledge gained from the architecture to help application engineers solve problems

- Define the three major steps in generating a solution

# Solution Generation
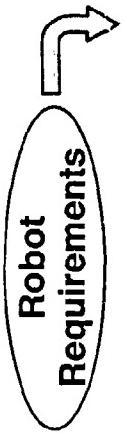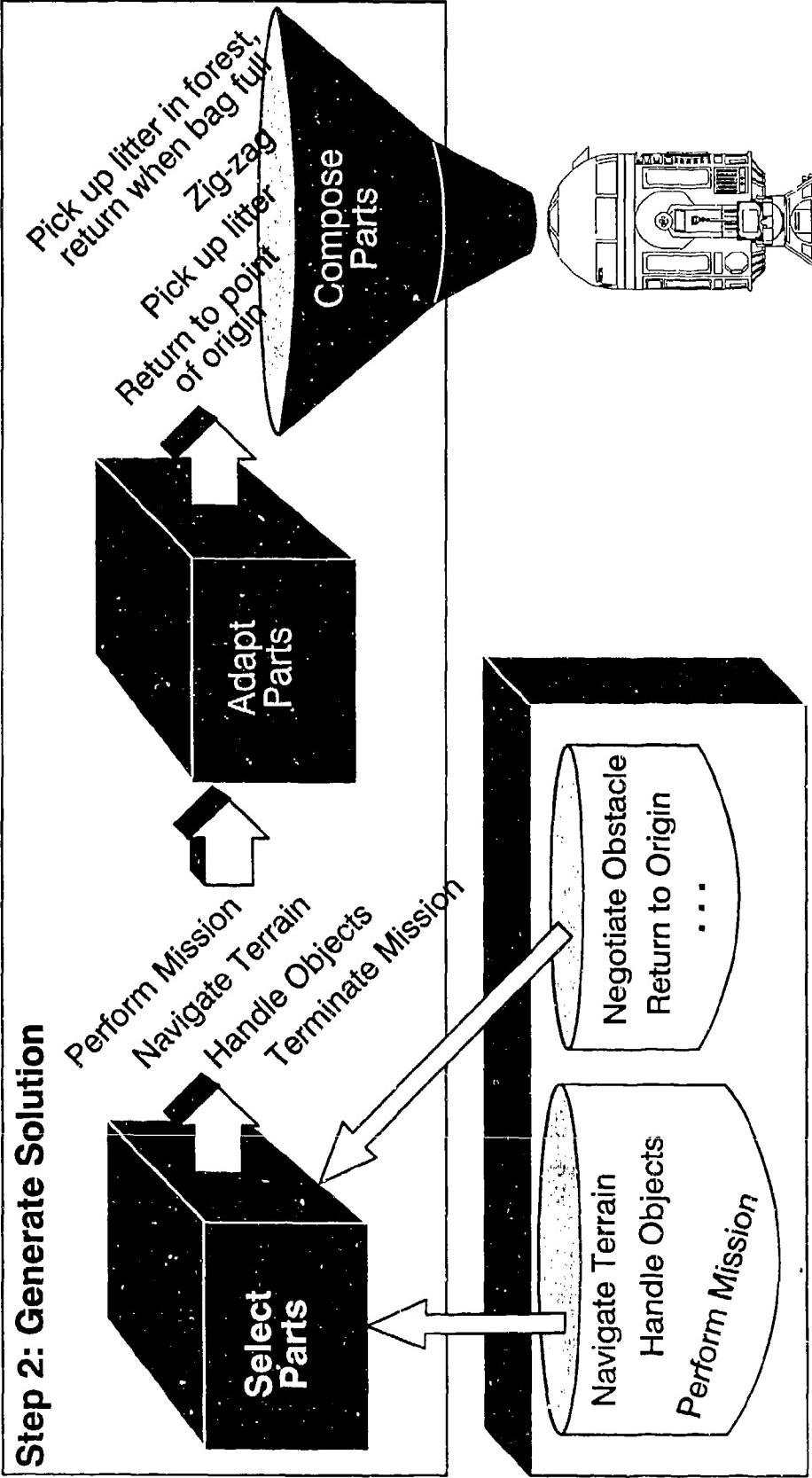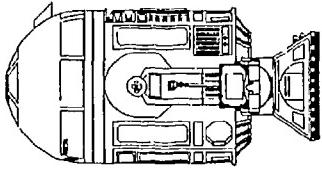
**Robot Requirements** →

## Step 2: Generate Solution

Select Parts

Perform Mission
Navigate Terrain
Handle Objects
Terminate Mission

Adapt Parts

Pick up litter in forest, return when bag full

Zig-zag

Pick up litter
Return to point of origin

Compose Parts

Negotiate Obstacle
Return to Origin
. . .

Navigate Terrain
Handle Objects
Perform Mission

**Reusable Software Components**

## DISCUSSION

This slide places Unit 4 in context by presenting the picture of megaprogramming. This picture shows the relationship between domain engineering and application engineering.

This unit has discussed everything above the dashed line. Domain engineering results in an understanding of the problems and solutions in a domain. This understanding can be used to develop:

- A process that helps application engineers precisely state a problem.

- A systematic means to generate a solution to a problem. It is based on an understanding of what software components make up a solution (the software architecture) and the procedures for using those components, and the procedures for using those components (the generation procedure). Once the domain engineer implements those components, they can be used and reused to solve any problem in the domain.

## STUDENT INTERACTIONS

- Now that you know a little more about domain engineering, would you rather be an application engineer or a domain engineer? Why?

- How would you define "megaprogramming"?

## OBJECTIVE

The students should be able to:

- Understand the basic functions of domain engineering and application engineering and the relationship between them.

# Megaprogramming

**Domain Engineering**

Understand all aspects of the domain (both problems and solutions)

Software and documentation to help solve a problem in the domain

Feedback

**Application Engineering**

Create a solution to a problem in the domain

The Solution

Your Customer

Customer Requirements

*Virginia*
*CENTER of*
*EXCELLENCE*
*for Software Reuse and Technology Transfer*

A-12

Overview of Megaprogramming Course: Unit 4, Domain Engineering, Lecture

# UNIT 4: DOMAIN ENGINEERING

## SUMMARY

Domain engineers are responsible for building what the application engineers need to develop solutions. This includes:

- Defining what is in the domain

- Defining the process that the application engineer will follow

- Developing process support (including reusable components) that the application engineer will use to state the problem, validate it, and generate the solution



**Process Support**
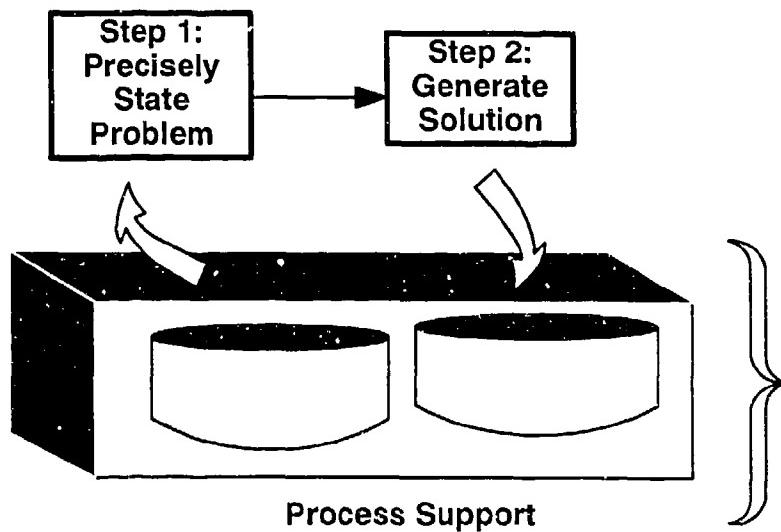
This is the "black box" support that the domain engineers create for the application engineers.

### DEFINING THE DOMAIN

Domain engineers decide what is in a domain.

Application engineers create individual systems. Domain engineers decide the range of systems application engineers can create.

Deciding what is in a domain involves studying the factors that constrain the problems and solutions which form the domain and deciding what problems and solutions are important.

Once the domain engineers know the problems that will be in the domain, they can study them and uncover:

- The commonalities among all problems

- The differences between instances of problems

### DEFINING THE PROCESS

An application engineer needs to know what steps to follow in order to develop a solution.

Domain engineers define the process for generating a solution in the domain and develop process support programs to help the application engineer. These support programs include:

- Support for defining and validating the requirements for the solution

- Support for generating the solution

## ARCHITECTURES

Every software solution is composed of components (e.g., procedures and functions). Every software solution has an *architecture*, which defines how the components work together.

Domain engineers create a "domain architecture" that:

- Defines the complete set of components used by all solutions in the domain

- Shows what components and interrelationships all solutions have in common

- Shows how individual solutions differ.

The following figure shows the domain architecture for the robot domain.



Domain engineers create these components. They also identify which components are common to all solutions in the domain and which are needed to solve specific problems.

# UNIT 4: DOMAIN ENGINEERING

## IN-CLASS DISCUSSION

1. Compare results of the laboratory activity. Is there more than one robot software architecture that satisfies the needs of each client? Why or why not?

2. What other kinds of robots could be produced by the URW, domain?

## HOMEWORK

1. Considering the domain of the URW, would you, as Chairman of the Board, want to produce robots to:

   a. Plant corn

   b. Pick water lilies

   c. Feed incubator babies

   In making your decision, are there enough similarities to warrant asking your domain engineers to write additional instructions?

2. The instructions in the left column were used to implement the software for a robot that searches a tundra for lost hikers. Each instruction in the left column is an adaptation of an architectural part in the right column. Match each instruction in the left column with the architectural part in the right column.

| Instructions | Architectural Parts |
|---|---|
| 1. Advance-north-moving-east-to-avoid-rocks-returning-when-bag-full | A. Perform Mission |
| | B. Navigate Terrain |
| Move north one unit. If a rock blocks the path, move east around it. If a hiker is found, pick him or her up; if doing so brings the robot to its full capacity, quit this instruction. | C. Negotiate Obstacle |
| | D. Handle Object |
| | E. Terminate Mission |
| 2. Advance-north-moving-west-to-avoid-rocks-returning-when-bag-full | |
| Same as Instruction 1, except that if a rock blocks the path, move west around it. | |
| 3. Sweep-east-returning-when-bag-full | |
| Move in a straight eastward line from the current position to the eastern boundary of the area to be searched. If a hiker is found, | |

pick him or her up; if doing so brings the robot to its full capacity, quit this instruction.

4. Sweep-west-returning-when-bag-full

   Same as Instruction 3, except move in a straight westward line from the current position to the western boundary of the area to be searched.

5. Sweep-south

   Move in a straight southward line from the current position to the southern boundary of the area to be searched. Ignore any hikers.

6. Sweep-west

   Same as Instruction 5, except move in a straight westward line from the current position to the western boundary of the area to be searched.

7. Pick-up-any-objects

   Pick up as many hikers at the current location as the capacity of the robot allows.

8. Return-when-bag-full

   Search tundra, looking for hikers. When the robot's capacity of hikers has been picked up, or when the entire tundra has been searched, return to the point of origin and turn off.

9. Return-to-starting-point

   From the current position, return to the point of origin.

10. Negotiate-rock-to-east-returning-when-bag-full

    Assumes that there is a rock just ahead of the robot, to the east. Moves the robot such that, when the instruction ends, the robot is just to the east of the rock, at the same latitude as when it started. If any hikers are found while negotiating the rock, they are picked up. If doing so brings the robot to its capacity, the instruction terminates, whether or not the rock has been negotiated.

11. Negotiate-rock-to-west-returning-when-bag-full

Same as Instruction 10, except assumes that there is a rock just ahead of the robot, to the **west**. Moves the robot such that, when the instruction ends, the robot is just to the **west** of the rock, at the same latitude as when it started.

12. Zig-zag-southwest

From the current position, zigzag southwest until reaching the southern or western boundary of the area being searched, whichever occurs first. Ignore any hikers.

3. URW, has been approached by the U. S. State Department. The State Department is concerned because it has received reports that embassies around the world have electronic bugs embedded in their walls. The State Department wants to know if URW can supply a robot that can locate these bugs. Fortunately, URW's engineers have just finished developing a new sensor, and they think it can be used for finding bugs. URW therefore decides to modify its robot domain so it can produce this new type of robot in addition to those in its old product line.

    a. For each of the following decisions in the decision-making process, state a requirement for the robot:

        (1)  Terrain

        (2)  Object type

        (3)  Choose if objects are to be carried

        (4)  Ending position

        (5)  Carrying capacity

    b. Identify the decisions from (1) through (5) whose range of allowed values must be changed to accommodate the new robot.

(Optional)

    c. Draw the software architecture for the robot, using the domain architecture as a starting point.

    d. Name some instructions from Question 2 that you think could be used without modification.

    e. Name some instructions from Question 2 that could be used with modification. What do you think the modifications might be?

    f. As a domain engineer for URW, what new components, if any, do you think would be necessary?

*This page intentionally left blank.*

# UNIT 4: DOMAIN ENGINEERING

## TEACHER NOTES FOR IN-CLASS DISCUSSION

1. Compare results of the laboratory activity. Is there more than one robot software architecture that satisfies the needs of each client? Why or why not?

2. What other kinds of robots could be produced by the URW domain? *This is really an open-ended question and should produce an interesting discussion.*

## TEACHER NOTES FOR HOMEWORK

1. Considering the domain of the URW, would you, as Chairman of the Board, want to produce robots to:

   a. Plant corn

   b. Pick water lilies

   c. Feed incubator babies

   In making your decision, are there enough similarities to warrant asking your domain engineers to write additional instructions?

*Note to teachers: Familiarity with Karel the Robot is helpful on the following questions.*

2. The instructions in the left column were used to implement the software for a robot that searches a tundra for lost hikers. Each instruction in the left column is an adaptation of an architectural part in the right column. Match each instruction in the left column with the architectural part in the right column.

<u>Instructions</u>                                    <u>Architectural Parts</u>

1. Advance-north-moving-east-to-avoid-rocks-       A. Perform Mission
   returning-when-bag-full                          B. Navigate Terrain
                                                    C. Negotiate Obstacle
   Move north one unit. If a rock blocks the        D. Handle Object
   path, move east around it. If a hiker is         E. Terminate Mission
   found, pick him or her up; if doing so brings
   the robot to its full capacity, quit this instruction.

2. Advance-north-moving-west-to-avoid-rocks-
   returning-when-bag-full

   Same as Instruction 1, except that if a rock
   blocks the path, move **west** around it.

3. Sweep-east-returning-when-bag-full

   Move in a straight eastward line from the current position to the eastern boundary of the area to be searched. If a hiker is found, pick him or her up; if doing so brings the robot to its full capacity, quit this instruction.

4. Sweep-west-returning-when-bag-full

   Same as Instruction 3, except move in a straight westward line from the current position to the western boundary of the area to be searched.

5. Sweep-south

   Move in a straight southward line from the current position to the southern boundary of the area to be searched. Ignore any hikers.

6. Sweep-west

   Same as Instruction 5, except move in a straight westward line from the current position to the western boundary of the area to be searched.

7. Pick-up-any-objects

   Pick up as many hikers at the current location as the capacity of the robot allows.

8. Return-when-bag-full

   Search tundra, looking for hikers. When the robot's capacity of hikers has been picked up, or when the entire tundra has been searched, return to the point of origin and turn off.

9. Return-to-starting-point

   From the current position, return to the point of origin.

10. Negotiate-rock-to-east-returning-when-bag-full

    Assumes that there is a rock just ahead of the robot, to the east. Moves the robot such that, when the instruction ends, the robot is just to the east of the rock, at the same latitude as when it started. If any hikers are found while negotiating the rock, they are picked up. If

doing so brings the robot to its capacity, the instruction terminates, whether or not the rock has been negotiated.

11. Negotiate-rock-to-west-returning-when-bag-full

    Same as Instruction 10, except assumes that there is a rock just ahead of the robot, to the **west**. Moves the robot such that, when the instruction ends, the robot is just to the **west** of the rock, at the same latitude as when it started.

12. Zig-zag-southwest

    From the current position, zigzag southwest until reaching the southern or western boundary of the area being searched, whichever occurs first. Ignore any hikers.

*Answers: 1—B, 2—B, 3—B, 4—B, 5—B, 6—B, 7—D, 8—A, 9—E, 10—C, 11—C, 12—B*

3.  URW has been approached by the U. S. State Department. The State Department is concerned because it has received reports that embassies around the world have electronic bugs embedded in their walls. The State Department wants to know if URW can supply a robot that can locate these bugs. Fortunately, URW's engineers have just finished developing a new sensor, and they think it can be used for finding bugs. URW therefore decides to modify its robot domain so it can produce this new type of robot in addition to those in its old product line.

    a.  For each of the following decisions in the decision-making process, state a requirement for the robot:

    (1) Terrain

    *Answer: The robot is to search buildings.*

    (2) Object type

    *Answer: The robot is to search for electronic bugs.*

    (3) Choose if objects are to be carried

    *Answer: The robot is to locate objects, but not carry them.*

    (4) Ending position

    *Valid answers: The robot is to stop when it locates a bug; the robot is to signal the location of each bug it finds, and continue until it has covered all of the building; or both. That is, URW should consider supplying both types of robots.*

(5) Carrying capacity

*Answer: The robot will not carry any objects.*

b. Identify the decisions from (1) through (5) whose range of allowed values must be changed to accommodate the new robot.

*Answer:  (1) — new terrain (buildings)*
         *(2) — new object type (bugs)*

(Optional)

c. Draw the software architecture for the robot, using the domain architecture as a starting point.

*Answer: Draw the architecture with particular nuances based on how the robot terminates its mission.*

d. Name some instructions from Question 2 that you think could be used without modification.

e. Name some instructions from Question 2 that could be used with modification. What do you think the modifications might be?

f. As a domain engineer for URW, what new components, if any, do you think would be necessary?

*Answer: The old search strategies do not work; however, realizing that is not simple.*

# Test for *Overview of Megaprogramming* Course

1. In the following table, check whether the task would be done by an application engineer or a domain engineer.

| Task | Application Engineer | Domain Engineer |
|---|---|---|
| Create the reusable components for a domain. | | |
| Work with the customer to understand the problem. | | |
| Validate the requirements. | | |
| Generate the solution. | | |
| Define what is in the domain. | | |
| Define the process and support needed to generate a solution for a customer. | | |
| Precisely state the problem. | | |

2. Read the following description of the Car4U Company:

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

**Have you ever wanted a car that was taller? wider? bigger? Have you ever shopped the car market and found nothing you wanted (and they still wanted a lot of money for it)? Well, no more, because now there's a new company for the discriminating buyer:**

### Do We Have a Car 4 U!

**The Car4U Company makes cars that are tailored to your every need and desire. You can have car seats that are tailored to your weight, height, and width. You can have bigger windows or smaller windows. You can have bigger trunks or smaller trunks. If you want four-wheel drive, you've got it. If you want your car to be a shade of blue that matches your eyes, we can do that too (in fact, we have over 1000 colors to choose from!). We have engines meant for cruising at high speeds and engines meant for climbing mountains. All in all, Car4U has over 3 dozen options. Each one is meant to help make your car truly your own.**

**We work with every customer to determine exactly what they want and then develop a car that suits their needs. No longer will you have to wait for the perfect car. Stop by your nearest Car4U store today and see what we can do for you!**

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

Based on this description, answer the following questions. Attach separate sheets if needed.

    a. What is the output of the Car4U Company's domain engineering activities?

    b. What is the output of their application engineering activities?

3. Read the following description of TJ's cash registers domain.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### Description of TJ's Cash Registers Domain

TJ's Cash Registers domain contains cash registers that can be used in just about any retail situation.

There are several options through which money can be entered into a cash register. The traditional way is to accept cash from the customer and store it in a removable money drawer. In addition to the money drawer, some cash registers are equipped with check imprinting services and/or the ability to scan in credit cards. In all cases, each cash register keeps track of the amount of money that has been received from the customer.

Several retail situations require the use of programmable keys that can store prices for items that are sold frequently. Other price input mechanisms include a price scanning function, a scale for items sold by weight, or the use of the numeric key pad. Only the numeric key pad and the programmable keys are standard, though the number of programmable keys can vary from register to register.

To show prices and to show other information for the cashier and the customer, each cash register has a digital display. Optionally, there may be a separate price display for the customer, either on the back of the register or on a completely separate, smaller display that is above the register and pointed towards the customer. After every transaction, each register automatically outputs a cash register receipt that is printed with the date and time.

Higher-end cash registers can be hooked up to the store's inventory system to either keep track of what the store has in stock (along with a warning message when the stock gets low) or to order items and have the customer pick them up at a separate location.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Based on this domain description, answer the following questions. Attach separate sheets if needed.

a. What are the members of the domain?

b. List the similarities between the members of the domain. Be specific.

c. List the differences between the members of the domain. Be specific.

# Survey for *Overview of Megaprogramming* Course

Please answer the following questions. The company that developed the course material will use this information to improve the course.

1.  Do you feel that you understand the basic principles of megaprogramming after taking this course?

2.  Do you see value in megaprogramming?

3.  Would you like to learn more?

4.  What activity(ies) or example(s) was most helpful to you in understanding megaprogramming?

5.  Do you have any other suggestions for how the course can be improved?

*This page intentionally left blank.*

# Test for *Overview of Megaprogramming* Course

# Teacher Answers

1.  In the following table, check whether the task would be done by an application engineer or a domain engineer.

| Task | Application Engineer | Domain Engineer |
|---|---|---|
| Create the reusable components for a domain. | | X |
| Work with the customer to understand the problem. | X | |
| Validate the requirements. | X | |
| Generate the solution. | X | |
| Define what is in the domain. | | X |
| Define the process and support needed to generate a solution for a customer. | | X |
| Precisely state the problem. | X | |

2.  Read the following description of the Car4U Company:

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

**Have you ever wanted a car that was taller? wider? bigger? Have you ever shopped the car market and found nothing you wanted (and they still wanted a lot of money for it)? Well, no more, because now there's a new company for the discriminating buyer:**

**Do We Have a Car 4 U!**

**The Car4U Company makes cars that are tailored to your every need and desire. You can have car seats that are tailored to your weight, height, and width. You can have bigger windows or smaller windows. You can have bigger trunks or smaller trunks. If you want four-wheel drive, you've got it. If you want your car to be a shade of blue that matches your eyes, we can do that too (in fact, we have over 1000 colors to choose from!). We have engines meant for cruising at high speeds and engines meant for climbing mountains. All in all, Car4U has over 3 dozen options. Each one is meant to help make your car truly your own.**

**We work with every customer to determine exactly what they want and then develop a car that suits their needs. No longer will you have to wait for the perfect car. Stop by your nearest Car4U store today and see what we can do for you!**

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

Based on this description, answer the following questions. Attach separate sheets if needed.

a. What is the output of the Car4U Company's domain engineering activities?

*Domain engineering would (1) create all of the different car components that would be needed to make a car, (2) create the ordered list of questions that the car salesperson would ask the customer, and (3) create the instructions for how the actual car builders would put together the car based on the specific needs of a specific customer.*

b. What is the output of their application engineering activities?

*Application engineering would (1) talk with the customer to understand what the customer wanted in a car, (2) use that understanding to come up with a precise statement of what was needed in the car, (3) make sure that this precise statement was what the customer wanted, and (4) generate the car (with help from the actual car builders) that met the customer's specific need.*

3. Read the following description of TJ's cash registers domain.

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

### Description of TJ's Cash Registers Domain

**TJ Inc. makes cash registers that can be used in just about any retail situation.**

**There are several options through which money can be entered into a cash register. The traditional way is to accept cash from the customer and store it in a removable money drawer. In addition to the money drawer, some cash registers are equipped with check imprinting services and/or the ability to scan in credit cards. In all cases, each cash register keeps track of the amount of money that has been received from the customer.**

**Several retail situations require the use of programmable keys that can store prices for items that are sold frequently. Other price input mechanisms include a price scanning function, a scale for items sold by weight, or the use of the numeric key pad. Only the numeric key pad and the programmable keys are standard, though the number of programmable keys can vary from register to register.**

**To show prices and to show other information for the cashier and the customer, each cash register has a digital display. Optionally, there may be a separate price display for the customer, either on the back of the register or on a completely separate, smaller display that is above the register and pointed towards the customer. After every transaction, each register automatically outputs a cash register receipt that is printed with the date and time.**

**Higher-end cash registers can be hooked up to the store's inventory system to either keep track of what the store has in stock (along with a warning message when the stock gets low) or to order items and have the customer pick them up at a separate location.**

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

Based on this domain description, answer the following questions. Attach separate sheets if needed.

a. What are the members of the domain?

*The members of TJ's Cash Registers domain are cash registers that could be built by TJ Inc.*

b. List the similarities between the members of the domain. Be specific.

*(1) Removable money drawer*
*(2) Ability to keep track of the amount of money received by customers*
*(3) Numeric key pad*
*(4) Existence of programmable keys*
*(5) Digital display*
*(6) Ability to output a cash register receipt*

c. List the differences between the members of the domain. Be specific.

*(1) Check imprinting services*
*(2) Ability to scan in credit cards*
*(3) Price scanning function*
*(4) Scale for items sold by weight*
*(5) Number of programmable keys*
*(6) Price display on back of register*
*(7) Separate price display pointed towards the customer*
*(8) Hook-up to store's inventory system to keep track of what's in stock*
*(9) Hook-up to store's inventory system to order items to be picked up at separate location*

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

# Survey for *Overview of Megaprogramming* Course

# Teacher Answers

There are no right or wrong answers on this section. A suggestion for this survey would be to hand it to the students after they have completed the test and give them extra credit if they fill it out and hand it in the next day.

*This page intentionally left blank.*

# List of Abbreviations and Acronymns

4GL     fourth-generation language

SGA     Student Government Association

3GL     third-generation language

URW     United Robot Workers, Inc.

*This page intentionally left blank.*

Abb-2

# Teacher Notes
## for
## Overview of Megaprogramming Course

# Teacher Notes
# for
# Overview of Megaprogramming Course

## SPC-94044-CMC

## Version 01.00.03

## September 1994

# CONTENTS

# FIGURES

# TABLES

# PREFACE

The Software Productivity Consortium (Consortium) is a consortium of aerospace companies that employ many of today and tomorrow's software developers. These industries have a strong interest in the quality of the software education today's youth receive, for a well-trained engineer is a valuable asset. The Consortium has therefore begun an ambitious program to infuse modern software development concepts into the computing curriculum. This program, run by the Megaprogramming Curriculum Project, is working with high schools and universities to devise new and innovative curricula and supporting materials.

The project's first product is the *Overview of Megaprogramming Course*. As of this writing, high schools in Virginia and West Virginia have incorporated the course into their school year. Based on reactions from those schools, the Consortium believes that the course provides an excellent introduction to megaprogramming (a foundation for many important software concepts, as the report will explain) and is useful to teachers who wish to keep their computer science courses in step with the state of the art.

The Consortium's original model for introducing the course at a high school was to provide personal instruction and consultation for teachers who expressed an interest in it. This approach is becoming impractical as use of the course expands. In any event, even one-on-one instruction runs the risk of omitting information. Hence this report, which captures the concepts covered during a typical tutorial session. It is not entirely a substitute for face-to-face contact, but it should help the reader understand megaprogramming. Moreover, it should convince the reader of the need for teaching the course.

The theme of this paper is that education in software development has for too long stressed programming and computer science at the expense of engineering. Industry wants software engineers, not computer scientists. Yet software engineering is still an optional course for most undergraduate computer science majors and is seldom, if ever, mentioned in high schools. To be sure, teaching engineering requires a scientific basis, and developing software is ultimately about programming; both topics are important. But to stress them at the expense of software engineering keeps the student from learning the full truth about why industry considers developing software to be so hard.

The Megaprogramming Curriculum Project hopes educators will agree that megaprogramming deserves a place in a student's education. To the degree that it can, the project actively seeks to work with schools in instituting megaprogramming, in soliciting feedback on the course, and in helping instructors revise and expand the megaprogramming curriculum material.

*This page intentionally left blank.*

# ACKNOWLEDGMENTS

*This page intentionally left blank.*

# 1. INTRODUCTION

This report is part of the Software Productivity Consortium's *Overview of Megaprogramming Course*. The course, aimed at high school and freshman undergraduate computer science students, teaches industrial software development concepts. It gives students a realistic look at how professionals build software. It covers important, practical issues often absent from today's classes. Although the course is brief (1 to 2 weeks), it helps students relate their programming knowledge to the real world.

The *Overview of Megaprogramming Course* focuses on software developed by the technique known as **megaprogramming**. The megaprogramming technique is very different from how most companies (and students) develop software today. Students develop a single program in response to a homework assignment; a company develops a single program in response to a business opportunity. By contrast, people using megaprogramming develop a whole **product line**—that is, a set of programs. In doing so, they use sound engineering principles that give them a solid understanding of each program's properties. This understanding may not help a company as it pursues a single business opportunity, but companies seldom pursue single opportunities; they go after sets of opportunities. Megaprogramming helps companies position themselves to obtain sets of opportunities. As this report will argue, it can also be helpful to students.

The development of the megaprogramming technique has been sponsored by the Advanced Research Projects Agency (ARPA) to help increase software's quality and decrease its cost (Boehm and Scherlis 1992). If megaprogramming is widely adopted, it will have a profound influence on how people build software. The difference will be as dramatic as when people first switched from assembly languages to FORTRAN or Pascal.

This report provides an introduction to megaprogramming. It is intended for anyone wanting to learn enough about megaprogramming to teach the *Overview of Megaprogramming Course*. It is one part of the materials distributed with the course and is best read in conjunction with the other materials. It does not assume familiarity with them, but it references the lecture notes, slides, and laboratory material. This may prove helpful to an instructor preparing lectures: a reference to a slide (e.g., "See Slide 1-4") generally indicates additional material that can be discussed when presenting that slide. Also, for each slide, Appendix A shows the material in this report most relevant to that slide.

This report is not to be viewed as a complete definition of megaprogramming. Rather, it presents megaprogramming's fundamental concepts and shows why these concepts are important in software development.

Moreover, this report argues that megaprogramming should be a basic part of a student's education in computing. The *Overview of Megaprogramming Course* is a first step in this direction. The course was created based on the belief that the computing curriculum has several significant deficiencies. The next several sections (1.1 through 1.3) address this point.

## 1.1 OVERVIEW OF THE CURRENT COMPUTER SCIENCE CURRICULUM

As of this writing, computer science education is a little over three decades old.[1] Those years have seen considerable change in how people develop software. Once it was largely an individual or small team activity that generally yielded programs of under 50,000 lines of code—small in today's terms. This is no longer the case. Advances in hardware have resulted in huge increases in available memory which, in turn, have led to larger programs. Today, large organizations routinely write programs containing millions of lines of code. They spend several years doing so and set up a complex hierarchy to build and maintain their products. Individuals and small teams still exist, but they work in ways that were inconceivable 30 years ago. They rely on a complicated software infrastructure of compilers, operating systems, editors, graphics libraries, and the like.

Computer science education has changed little in this period. The standard computer science curriculum, now and then, begins by introducing students to programming. Students learn the syntax and semantics of a computer programming language such as Pascal or Ada and are taught some rudimentary concepts of formulating algorithms to solve problems and verifying these algorithms by creating and executing test cases. They apply this knowledge to formulate algorithms in some high-level programming language, which they then compile, execute, and test. This fills up one course; subsequent courses introduce such topics as the science of algorithm analysis, the design and implementation of operating systems, or—often only in a student's final undergraduate year—a discussion of software engineering, that is, how professionals build software.

This curriculum model, shown in Figure 1, makes students reasonably adept at solving simple problems after only one course. This is commendable, but it has two significant disadvantages:

- *It introduces students to software development before teaching them any science they might use to analyze the quality of their software.* Real software must do more than simply compute the right answer. It must execute efficiently. Its interface must be friendly to its users. It must integrate smoothly with the system of which it is a part. Other people must be able to understand it. In the hard sciences, such as chemistry or physics, students immediately learn quantitative analysis techniques to address issues analogous to these. Quantitative and qualitative analysis techniques exist for software, but students seldom learn them in their first course. As a result, students learn to see software development as an art or craft, whereas it should be an engineering activity—an application of scientific principles.

```
┌──────────────┐     ┌──────────────┐     ┌──────────────┐     ┌──────────────┐
│    Learn     │     │    Learn     │     │    Learn     │     │    Learn     │
│ Programming  │ ──▶ │   Computer   │ ──▶ │   Special    │ ──▶ │   Software   │
│   Language   │     │   Science    │     │    Topics    │     │  Engineering │
└──────────────┘     └──────────────┘     └──────────────┘     └──────────────┘
```
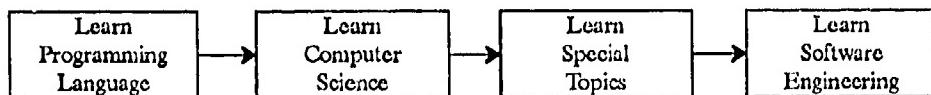
Figure 1. The Current Curriculum Model for Teaching Computing

- *The current curriculum model focuses on programming, a very small portion of the software development process* (see Section 2). The programming part is most amenable to concrete analysis and requires the least abstract thought. However, as Section 1.2 discusses, most fundamental skills a software developer needs are not part of programming per se. These skills are not usually taught until the software engineering course. Instead, students learn workaround techniques that are of questionable value to a professional (Frey, Cohoon, and

---

1.  Purdue University founded the United States' first computer science department in 1962.

Fife 1994). By the time they take a software engineering course, the undesirable techniques are firmly entrenched in their minds.

## 1.2 DEFINITION OF TERMS

This report so far has discussed "building" and "developing" software, and might also have included commonly used words like "create," "produce," and "write." Students earn degrees in Computer Science. They take courses with titles like Introduction to Programming and Software Engineering. They use the skills they learn to develop software. What, exactly, is the relationship between all these terms?

Answering this question requires a combination of historical, academic, and industrial perspectives. Industry's goal is to develop software. Hence, the phrase **software development** refers to the whole process of creating and rewriting software, starting with the first realization of its need, on through its first use, and from there through bug fixes and enhancements to the time when it becomes obsolete and is finally discarded (see Slide 1-4). People use this and words like build, develop, write, and create interchangeably—although Brooks (1987) reports an epiphany on realizing their distinction. "I still remember the jolt I felt in 1958 when I first heard a friend talk about *building* a program, as opposed to *writing* one," he writes. "In a flash he broadened my whole view of the software process."

Software development, the previous paragraph claims, is a **process**. What does this imply? Webster's Dictionary (Merriam 1977) defines a process as "a series of actions conducted to an end." An organization can perform a process in many ways; many are chaotic and poorly understood. These are the bane of an organization, for poorly understood actions are hard to plan and manage. Organizations want their processes to introduce **engineering**. In engineering (to paraphrase Webster's), science is applied to processes to make them useful to humanity. In other words, software development that incorporates engineering has a scientific basis. This science helps organizations predict the properties of software without actually building it—just as a civil engineer uses the science of structural mechanics to predict the load a bridge can hold without actually building it. Similarly, science also helps organizations create and perform a software process, just as chemistry helps a chemical engineer create and perform a process to manufacture chemicals.

**Software engineering**, then, is software development using engineering. The science underlying this engineering comes in part from **computer science**. Computer science lets software engineers predict many important properties of software. For example, algorithm analysis lets them know that sorting $n$ elements takes time proportional to $n \ln n$. Computer science also provides standard algorithms and techniques. Backus (1978) reports that writing the first high-level language compiler took 3 years. Today, thanks to research in areas like formal languages and parsing, students taking a compiler class learn a well-defined science that lets them complete a compiler in a single semester.

Computer science is not the only science needed in software engineering. Large teams—sometimes thousands of people—develop software. Managing these teams is a complex social process, and so the social sciences, notably psychology, have made important contributions to software engineering; Weinberg (1971) is an excellent example. Business sciences, too, have contributed their share, tailoring management theories to software (Humphrey 1989) and providing models for estimating the costs of developing software (Boehm 1981).

This broad focus distinguishes software engineering from **programming**. Programming is usually taken to mean the parts of software development concerned only with writing and testing programs. Programming courses do not, as a rule, teach science. Nor do they teach a process of software

development, engineering-based or otherwise. Students who only learn programming may know how to create programs, but they do not know how to assess them. They also do not appreciate the multidisciplinary nature of developing software.

In a famous debate at the annual Computer Science Conference (CACM 1989), Edsger Dijkstra argued that software engineering is not really engineering at all. He claimed computer science is still too young to provide enough science for software development to be an engineering discipline like civil or electrical engineering. Even his opponents accepted his opinion, at least in part. They realize that "software development" more accurately describes what goes on in industry today than does "software engineering." This, however, does not imply people should avoid teaching what software science and engineering is known. Moreover, software is developed in response to a problem in some area such as civil or electrical engineering. The science from these areas can and should be put to use during software development. Computer science courses stress this point all too infrequently.

## 1.3 PROBLEMS WITH THE CURRENT CURRICULUM

The discussion in the previous sections leads to the following description of specific problems with the current curriculum:

- *The initial emphasis is on programming,* widely regarded as the easiest part of software development (Ince 1988). Certainly, mastering programming requires grasping many new language and logic concepts, and is not a trivial activity. But neither are any of the other aspects of software development, and students deserve to learn about them as well.

- *Science and engineering skills are played down.* Students spend so much time learning programming, they forget that programs are written to achieve an end. Professional software developers, when they build a program, must be facile in areas other than programming. If they are writing satellite control software, they must understand equations of satellite motion. If they are writing an accounting package, they must understand business science. The computing curriculum, by contrast, emphasizes programming and computer science concepts without asking students to apply other sciences.

- *Students are discouraged from using existing software.* Professionals strive to avoid writing code if they or someone else have written it before. They try to reuse existing code. Doing so is not always easy, but the savings in time and effort is often immense. Students do not learn reuse techniques such as megaprogramming and are often informed that using somebody else's code is plagiarism. Plagiarism aside, the result is that students must reenter code they have written before—a time-wasting, rote activity. From a pedagogic perspective, students learn computer science but not software engineering. They see computer science concepts building on previously learned computer science concepts (for example, analysis of sorting routine execution time builds on the science of algorithm analysis), but they do not see engineering concepts building on other engineering concepts.

## 1.4 THE MEGAPROGRAMMING CURRICULUM PROJECT

Addressing the problems with the current curriculum is the objective of the Megaprogramming Curriculum Project. This project, founded in 1992 by the Software Productivity Consortium (the Consortium), has the long-range goal to create, foster, and encourage curricula for high schools and universities that include more software engineering.

The project is accomplishing its long-term goals, in part, by creating short courses. Instructors use these courses as special topics units that introduce students to software engineering concepts. Teachers also gain knowledge of software engineering. They incorporate this knowledge into their regular courses. The Megaprogramming Curriculum Project's *Overview of Megaprogramming Course* is a broad overview of modern software engineering.

## 1.5 PURPOSE OF THIS REPORT

This report serves the following purposes:

- *The report complements the lecture notes for the* **Overview of Megaprogramming Course.** The notes are organized as slides suitable for making into transparencies; each slide has an accompanying page of explanatory notes. This organization, suited to a lecture, necessarily abridges the notes. This report provides information that could not fit into the notes. Sometimes this information provides context; while not likely to be incorporated into a lecture, it shows the importance of megaprogramming in industry and academia. Other times, the information simply did not fit into the flow of the lecture.

- *The report provides instructors with answers to questions perceptive students might ask.* This information has often been omitted from the slides because it cannot fit into the format. Indeed, presenting all of it would result in a course much longer than the anticipated 1 to 2 weeks. However, instructors will want to be prepared to go into depth on certain topics when students show interest.

- *The report increases awareness of industrial practices in the educational community.* Instructors can use this knowledge to teach practical aspects of software development, introduce realistic concerns, and create projects and assignments incorporating more real-world considerations.

- *The report assists instructors in preparing their own megaprogramming lectures, examples, and exercises.* The *Overview of Megaprogramming Course* cannot possibly suit everyone's style. The Megaprogramming Curriculum Project encourages instructors to develop their own materials. This has been its model since the very first year, since curriculum propagation on a nationwide scale is beyond the scope of a single project.

## 1.6 AUDIENCE FOR THIS REPORT

This report is intended primarily for high school and university instructors who are interested in teaching the *Overview of Megaprogramming Course*. It is also of interest to anyone who wants to know about the Megaprogramming Curriculum Project and its tenets. The reader should understand enough programming and basic computer science concepts to teach an introductory course.

## 1.7 ORGANIZATION OF THIS PAPER

The material in this report is organized as follows:

- Section 1 contains an overview of the current computer science curriculum, definitions of basic terms, problems in the curriculum, and a description of the Megaprogramming Curriculum Project and how it addresses those problems.

- Section 2 discusses a hypothetical software project, intending to present a reasonable picture of the current state of industrial software development.

- Section 3 discusses the government's concern about the growing costs of software development, and the Megaprogramming Effort, which is responsible for starting the Megaprogramming Curriculum Project.

- Section 4 covers megaprogramming in detail: what it is and why practicing it seems likely to improve the state of software development.

- Section 5 discusses the importance of megaprogramming in academia, arguing for adoption of the ideas of the Megaprogramming Curriculum Project into high school and college curricula.

- Appendix A delineates how the lecture material relates to the organization of this report.

- Appendix B discusses the structure of the *Overview of Megaprogramming Course*.

## 1.8 TYPOGRAPHIC CONVENTIONS

This report uses the following typographic conventions:

Serif font ...................... General presentation of information.

*Italicized serif* font .............. Mathematical expressions and publication titles.

**Boldfaced serif** font ............... Section headings and emphasis.

***Boldfaced italicized serif*** font ....... Run-in headings in bulleted lists.

# 2. SOFTWARE DEVELOPMENT IN INDUSTRY TODAY

Discussing the importance of megaprogramming or the need for curriculum changes requires an understanding of today's software development practices. This section describes issues that companies in the software business face. It goes well beyond software, for software is a means to an end and is influenced and constrained by a variety of forces that at first seem only peripherally related. Yet these forces shape the software throughout its useful lifetime. Therefore, understanding them is important.

The lectures and laboratory in the *Overview of Megaprogramming Course* refer to United Robot Workers, Inc. (URW), a fictitious company in the robot business. How does URW conceive the need for a particular robot model? How does it turn that need into a working product?

## 2.1 EXAMPLES OF SOFTWARE PROJECTS

The answer to these questions is that URW follows a **software development process.** In software engineering, a process is best thought of as a set of activities carried out in some predefined order. An algorithm is one example of a process. An algorithm is a very precisely defined process, sufficiently precise that a computer can perform it. The processes that companies use to build products are, as a rule, vague in describing how to perform each activity, when each activity can begin, and when it can end. But the process adds enough order for URW to pose and solve problems in a logical sequence.

Figure 2, drawn from Slide 1-5, shows a software development process URW might follow to build a robot. Each box represents an activity. Arrows between the boxes show the order in which the activities must be performed: an activity at the head of an arrow begins after the activity at the arrow's tail ends. The labels on the arrows are the products that result from each activity.

This process begins with the **Requirements** activity. Here, URW's engineers determine the problem they are trying to solve: Who is their customer, and what is he or she trying to accomplish? In other words, what characteristics must a solution to the problem possess? Answers to these questions let URW know what type of robot to build. For example, the laboratory at the end of Unit 3 (see Appendix B) asks students to consider several different customers. One is a farmer who wants to harvest corn. Another is a representative from the National Park Service, who wants to pick up litter. The characteristics of a solution for the farmer are quite different from those for the National Park Service:

- The general shape of the robots will differ, because of the terrains. A robot in a cornfield does not have to contend with closely packed trees. Furthermore, it must carry large amounts of corn—hundreds or even thousands of ears. By contrast, a robot that picks up litter in a forest will zigzag around trees, limiting its size; this small size means a small carrying container, so it can carry a much smaller volume of material than the cornfield robot. These factors have implications on the motor and power technologies too.

- The robots will need different software. The litter-carrying robot encounters trees, and so needs obstacle-avoidance routines. The cornfield robot does not.
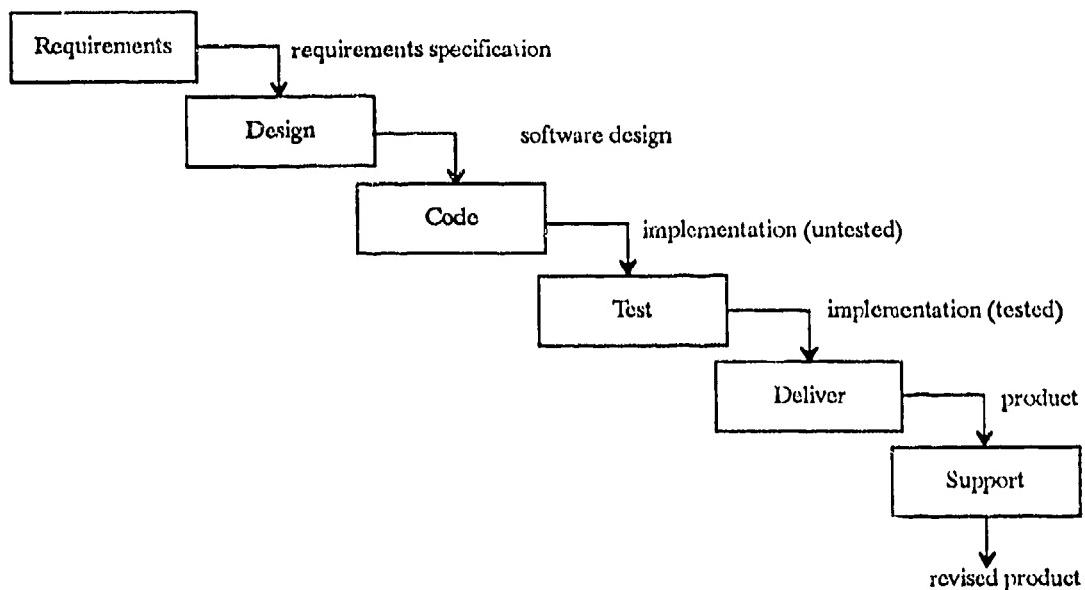


Figure 2. Software Development Process

The type of customer introduces some interesting differences. The National Park Service is part of the United States Government, and the government does not buy things the same way as a farmer. The farmer purchases the robot at a store. The store's sales staff show her a complete product line of robots, and try to convince her that one of their robots will meet her needs. If she agrees, she buys the robot. If not, she either goes to a competing store or decides to do without a robot. If she ends up purchasing a robot, she probably gets one that does almost everything she needs (but not quite) and has a few features she doesn't want (rather like a nonsmoker who buys a car: all cars come with ashtrays).

To satisfy this market of farmers, URW engages in **commercial software development**. URW will build a line of robots with the features they think farmers want; they won't anticipate everyone's needs perfectly, but they will produce a product that satisfies most farmers most of the time. URW anticipates that this strategy will yield high sales volume, justifying the extra costs of adding sometimes unused features to the robot and its software.

The government does not operate this way. Rather than selecting from among a set of ready made commercially available "off the shelf" robots, it will present its problem to a company and ask them to create a robot that solves its particular problem. URW will sign a contract promising to build a robot that solves exactly the government's problem—no more and no less. URW will agree to build the robot at a predetermined price and to deliver it within a stated time period. This is called **contractual software development**.

Software developed for the commercial market is very different from software developed under contract. Commercial software must appeal to a broad range of people; contractual software must appeal only to its purchaser. Commercial software is developed in anticipation of customers; contractual software is developed for a specific customer. Commercial software, then, tends to be broader in scope, with many features; contractual software, by comparison, is more focused.

Unexpectedly, commercial software usually contains fewer lines of code than contractual software. Although more focused, contractual software tends to solve problems that are inherently more complex than those tackled by commercial products. For example, word processing (commercial) is simply not as challenging as satellite control (contractual). The equations for arranging text are trivial compared to those for keeping a satellite in orbit and in touch with earth.

Furthermore, commercial and contractual software buyers have different expectations of their products. If the farmer's robot fails because of a software error, she will be justifiably upset, but she is not likely to suffer catastrophic loss. By contrast, failure of satellite software may render an already launched billion dollar satellite inoperable; failure of navigational software might result in hundreds of people losing their lives in an airplane crash. This explains why some commercial software corporations have a reputation of letting their users find errors in their products, whereas the government insists on extremely rigorous testing of its software (which still doesn't always find all of the errors). It also explains why one government study estimated that the cost of testing a satellite control program came to $1,000 **per line** of code. Lastly, it explains why some of the space shuttle's hardware and software dates from the Apollo program. The software may have been written long before anyone knew about structured programming, and the hardware may be antiquated, but both are known to work in situations where death is the price of failure. NASA's budget cannot stand the expense of rewriting and retesting the software for modern hardware.

The software is not the only thing that differs between contractual and commercial software development. The software development process itself differs considerably. Figure 2 is a reasonable abstraction of both cases, but the details of each activity bear closer scrutiny depending on the model in use. The following discussion (Sections 2.1.1 and 2.1.2) will concentrate first on the contractual model. It will then show how the process differs when URW uses the commercial model.

### 2.1.1 A CONTRACTUAL SOFTWARE DEVELOPMENT SCENARIO

Figure 2 shows that each activity of a process yields some product. The product that results from the Requirements activity is a **requirements specification**. This is a statement of the problem URW is to solve. The requirements specification comes from the customer, at least initially, because the customer is the one who recognizes the problem. In the contractual software development model, the customer will create a preliminary version and announce that he wants to award a contract. Companies such as URW will submit proposals on developing the software, giving a cost and schedule. The customer will select one (typically, the lowest bidder) as the contractor. The customer and the contractor will work together to refine the customer's preliminary version of the problem into a precise requirements specification. This specification serves as a contract between the customer and the contractor. It states exactly what software the contractor must develop to satisfy the customer.

What sort of information does a requirements specification contain? Figure 3 shows a sample table of contents, annotated to explain each section. The guiding rule is to define what the software must do, but to avoid stating how the software must do it. Slide 1-5 compares requirements specifications to homework assignments. This is a good analogy: the teacher presents students with a problem without divulging how to solve it.

Assume that URW submits the winning proposal. The requirements specification guides URW during the next activity: **Design**. During this activity, URW conceives and plans how the robot will work. This includes such issues as:

- What is the shape of the robot? During the requirements phase, URW will have noted the terrain in which the robot must operate, and that the terrain imposes constraints. During design, URW must choose a shape that works for the constraints.

- What hardware will be used to build the robot? For instance, what types of sensors will it have? What type of locomotion mechanism will it use? Furthermore, what tasks will the software perform? That is, what is done by hardware and what is done by software?

- What is the **software architecture**? Software has an architecture, just like a house. URW's engineers create a view of the robot control software as a set of interacting **components**. Each component will play a specific role. For more information on the architecture of the robot software, see the Unit 4 lecture.

**1. Introduction.** An overview of the problem: What is it, and—broadly speaking—what is the nature of the solution that will be proposed?

**2. Inputs and Outputs.** A description of how the system is connected to the outside world. For example, a word processor uses a keyboard and mouse as input and a screen and a printer as output. An automated teller machine uses a keypad and card reader as input, and a screen and a money dispenser as output. URW's robots use sensors and compasses for input and control arms and locomotion devices as outputs.

**3. Modes of Operation.** A description of the different operating modes, as a user might conceive them. For example, word processors often have text entering modes, ruler setting modes, equation modes, and table modes. Modes providing a means to categorize the software functions, simplifying their description.

**4. Description of Software Functions.** This is the meat of the requirements specification. It describes the legal inputs to the system and how the system must respond to each. This response is stated in terms of the outputs produced. It avoids mentioning algorithms (it is analogous to how a teacher tries to phrase homework assignments)

**5. Reactions to Undesirable Events.** The description of software functions describes responses to *legal* inputs. This describes responses to illegal inputs and other "undesirable events" (like detection of hardware problems). In a word processor, shutting down is usually acceptable. Software controlling a satellite hurtling toward the outer planets lacks this luxury; it must try to recover from an undesired event.

**6. Required Subsets.** Systems are often planned in full but built in stages. This section describes acceptable intermediate subsets.

**7. Glossary.** A description of terminology used in the requirements specification.

Figure 3. Sample Software Requirements Specification Table of Contents

Figure 2 shows the software development process as it appears in Unit 1. In fact, Slide 1-5 is a simplification of the design process. Usually, design consists of two activities: **Architectural Design** and **Detailed Design**, shown in Figure 4. The architectural design describes the software as a set of components. The detailed design describes the inner workings of each component. In other words, experience has shown that software design is easiest if one first splits the design into parts, then concentrates on the details of those parts. This is akin to an architect designing a house by laying out the rooms and their relative positions, then determining each room's details—the positions of the electrical outlets and ventilating grates, for instance. The overall layout of the major components (rooms) takes precedence over the details.

The Requirements activity was performed jointly by URW and its customer. URW's engineers are responsible for creating the design and do not expect the customer to be involved. As part of the contract, however, the customer will usually insist on a review after each activity. Thus, the customer will hold a **preliminary design review** after architectural design and a **critical design review** after

detailed design. These reviews will assure the customer that URW's progress is satisfactory and give URW a chance to get feedback on the quality of their design.
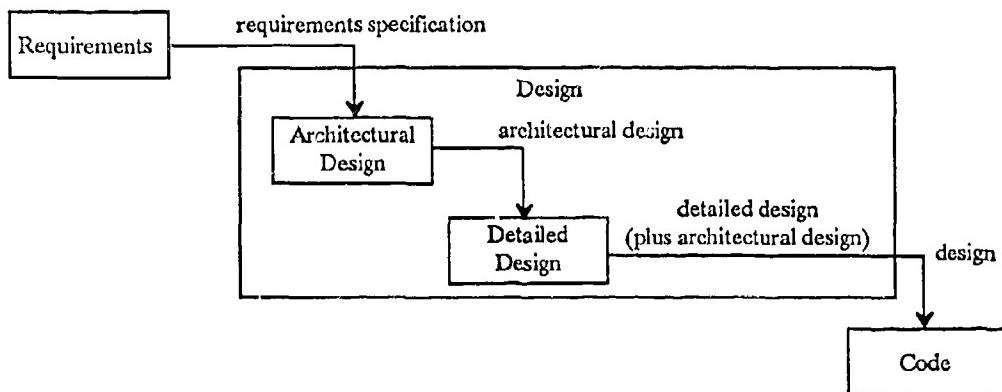


Figure 4. Activities of Software Design

Design is followed by the **Coding** activity. Here, URW's engineers implement the design by expressing it in a programming language. The design envisions a solution to the problem expressed by the requirements; the code realizes that vision, just as a house is built to blueprints. Unlike the design, the code can be compiled and executed. URW can use it to run a robot.

The Coding activity begins with the design and ends when URW's engineers have created a first, tested version of each component. In large software development projects, URW will divide its engineers into teams and make each team responsible for implementing a particular set of components. This way, the teams can work in parallel.

Once a team finishes a component, they begin the **Test** activity. Testing occurs in two parts: **unit testing** and **integration testing**. During unit testing, a team tests a component they have coded. They test the component as an indivisible unit, according to the information about that component in the detailed design. After a team has tested a component, they integrate it with other components that have passed unit testing. They subject the resulting subsystems to testing; in other words, they assemble the software component-by-component, until all components have been integrated. This process of integration and testing is referred to as the **Integration Testing** activity. When they have integrated all components and tested that resulting product, they have a working system.

URW must next deliver the software to its customer. Actually, since URW builds robots and not just software, it will integrate the software with the robot hardware.[2] It can then deliver the robot to the National Park Service. This involves transporting the robot to its destination, performing final tests in the actual environment, and training necessary personnel to operate it.

Finally, URW must support the National Park Service in using the robot. Despite URW's engineers' best efforts, the robot will probably fail occasionally because of errors URW made while designing and implementing the software. URW will be responsible for fixing these errors and for delivering the corrected software to the customer. It is also likely that the Park Service did not fully understand the scope of the problem and made errors in the requirements. For example, they might have forgotten that the trail they want the robot to patrol floods each spring; therefore, they did not ask for a watertight robot.

2.  URW will have a hardware development process as well as a software development process. The combined effort to develop hardware and software is called the **system development process**, but that's outside the scope of this paper.

Accordingly, they may ask URW to redesign its robot. URW will be asked to correct design errors at no charge, since such errors are its own fault; but the Park Service must pay URW to implement changes in the requirements, because the original requirements were a contract that URW fulfilled. Each change will be done by following the software development process in miniature, modifying existing work rather than creating products from scratch. The Park Service and URW will first agree on the modified requirements. URW will then determine how the change affects the design. It will then modify the code, test the modified version, and deliver the new version to the Park Service.

Error-free products are, unfortunately, the exception rather than the rule. This is true of all products, requirements and design as well as code. Datamation (1994) reported that California's Department of Motor Vehicles committed 28 million dollars to modernizing its computers. The new facilities were installed after 6 years, at a cost of 44.5 million dollars—and were junked after a few months, because the requirements had been expressed improperly. Ince (1988) describes a U.S. Government Accounting Office report showing that over 90% of government-contracted software systems were unusable as delivered—and of that 90%, over 50% had requirements so poorly conceived that they could never be used. The cost of modifying the software would have exceeded that of maintaining the status quo.

In other words, Figure 2 is something of an idealization. Royce (1970) termed it a "waterfall model" of software development, since it depicts work flowing smoothly down from one activity to another, as water flows downstream over a series of falls. In reality, the work flows upstream too: Figure 2 should show arrows from design to requirements, from code back to requirements and design, and so on—that is, from each downstream activity to all activities upstream from it. The arrows would denote errors from a previous activity caught during the activity at the arrow's tail. People omit these arrows to simplify the picture, claiming that the arrows shown in Figure 2 depict the most significant work flows. The data in Slide 1-6 contradict this, as do the arguments of many researchers in the area (McCracken and Jackson 1982).

## 2.1.2 A Commercial Software Development Scenario

When URW develops a robot to harvest corn in a field, it will follow the process in Figure 2. The essential steps of commercial software development remain the same as for contractual software development. However, the objectives of the activities are quite different. This section gives a scenario for commercial software development.

The Requirements activity differs most. There is no customer to develop the initial requirements specification for URW. Instead, URW must perceive the market for a corn harvesting robot and determine the characteristics of that robot themselves. This is a risky operation. URW should consult with farmers who may be potential customers, trying to understand what they would like in a robot—in fact, trying to understand if there really is a market for such a robot. Many a clever invention has failed because it solved the wrong problem, because it had too much close competition, or because people turned out to be satisfied with what they had.

The requirements specification that URW develops is therefore not a contract. It is only a description of the problem to be solved. As in contractual software development, URW's engineers use it to guide them during the Design, Code, and Test activities. These three activities do not differ significantly from their counterparts in contractual software development. URW's goal for each is still the same: to produce a product that is a solution to the problem stated in the requirements. However, there is no customer. An outsider does not fix a schedule; URW simply tries to get its product to market quickly. URW holds design reviews solely for its own benefit.

This isolation poses problems for URW. Software engineers have long recognized that the only reliable way to assess software's correctness is to try it in an environment representative of its ultimate market. If, during development, URW is beholden to no customer, how can it ensure that its products will satisfy customers? URW might address this risk by performing **alpha testing** and **beta testing**. URW "alpha tests" its product by using a preliminary version in a realistic environment. Engineers operate the product as if it were the final version, except no one is surprised when it fails. This provides URW with useful feedback without fear of alienating customers.

URW follows alpha testing with beta testing. During beta testing, URW gives the alpha-tested version of its product to a few selected customers. These customers, who are usually given some financial incentive, also operate the robot as if it were a final product, again with the realization that it will probably fail (URW selects customers who are potentially interested in the robot but whose business is not jeopardized by its failure). Customers report failures and other problems to URW. As URW receives their reports and corrects the problems, it gains confidence that its product is robust enough to compete in the marketplace.

In the Deliver activity, URW does not deliver the software to a customer. It markets its product through an appropriate channel, like a store or a mail-order service. In the Support activity, it gauges its product's success based on sales. It keeps abreast of interest in the product and determines whether the product can be improved—often, introducing a product changes the environment in which it is used, leading to new product opportunities. (Witness the automobile, whose capability for greater speed than the horse led to the paved road; in turn, paved roads led to faster automobiles.)

In short, URW is not directly responsible to a single customer, as in contractual software development. Although contractual software tends to be more complex than commercial software, the requirements for commercial software are much harder to discern and much more likely to change. Commercial companies usually introduce new products more frequently than contract-based companies. They often have a complete **product line**—that is, a set of related products—so they can compete in several market niches. Where the contractual company supports only a single version of a product, the commercial company must support each product in its product line—not to mention always having to consider whether to add new products to the line.

## 2.2 PROBLEMS WITH SOFTWARE DEVELOPMENT

Section 2.1 showed how URW could follow an organized software development process to create its robot control software. So what could go wrong? Plenty. In reality, software development is just this side of chaotic. This section shows some of the reasons why it is difficult.

- *Expressing software requirements.* Requirements written in English are usually ambiguous: the gift of clear expression is given to few. Requirements are often incomplete. One way to think of software requirements is that they should describe all possible inputs and all allowable responses to those inputs. No one has yet discovered a good method for determining whether requirements written in English can describe all the inputs and responses.

  Because of this problem, many people have proposed expressing requirements using mathematical notations. Such notations are unambiguous and can be checked for completeness—often by automated tools, which is especially helpful as it relieves people from performing a tedious, time-consuming chore. However, many people find these notations difficult to learn and use. No conclusive evidence exists that mathematical notations are more effective than using English.

Furthermore, the real problem with requirements is not so much whether they are complete and unambiguous, but whether they describe the right problem. Time and again, experience has shown that people have difficulty fully grasping a problem—and grasping the problem is vital to producing the correct solution. This is well illustrated by a radar system the U.S. built in Greenland in the 1950s to detect missiles over the North Pole. Early in its operation, it reported the approach of a missile the size of the moon—indeed, it was the moon, for the people who wrote the requirements had overlooked that celestial body's existence. The engineers who wrote the software from those requirements did their job perfectly, for they satisfied their contractual obligation to detect high-altitude, distant objects. Sometimes, the most obvious things are most evasive.

Determining whether the requirements describe the right problem is termed **validation**. Validation is different than **verification**—determining whether the software behaves according to the requirements. An organization can perform verification on its own, but cannot validate software except by exposing it to customers. (The laboratory illustrates this point: the Validate Requirements step forces students to put questions to their hypothetical customers.) Therefore, validation is risky. In contractual software development, customers may be forced to admit they did not understand their needs. In commercial software development, an organization faces embarrassment if its potential customers judge its product shoddy or ineffectual.

- *Following a software development process.* Telling someone they must write a software requirements specification is one thing. Telling them how to write that specification is another. Software developers understand the processes they must follow much better than they understand the **methods** for performing individual activities of a process. Despite extensive research in the area, specifying and designing software remains something of an art. People are told that an activity must yield a certain set of products, but are on their own as to how they should create those products. People have proposed methods, such as structured analysis (Marco and McGowan 1987) and object-oriented design (Coad and Yourdon 1990); but methods ease the problem at best. They do not solve it.

The discussion of Figure 2 on page 12 mentioned that it omits mistakes and therefore describes software development only partially. This is another dimension of the difficulty of following a process exactly. The picture is supposed to depict an orderly sequence of activities that occur one at a time. In reality, several occur simultaneously, working with partially finished and not fully correct products. Such a situation is very difficult to manage.

- *Keeping requirements and documentation up to date.* All too often, when errors are found in requirements or design documents, no one bothers to create a new, fixed version. Even though the errors are corrected, nobody records the change except in the software. The software is, after all, the goal. As long as it behaves as everyone expects, why bother correcting a mistake in the design document?

The answer is to avoid confusing the next person who reads the design document. Unfortunately, software developers are under pressure to get the software up and running. Every change they make to the design document delays the software. Those delays cost money, in the form of lost sales or broken contracts. Companies concentrate on the end product and ignore the intermediate ones. They forget one crucial detail: the majority of effort that goes into software occurs after the first version is deployed (Boehm 1981). One reason people

spend so much effort during that time is because they're reading incorrect and out-of-date requirements specifications and design documents. These documents were supposed to help them understand the software and facilitate its maintenance; instead, they confuse.

- *Grasping software design.* No one has discovered a way to describe software design clearly and concisely. Many techniques have been tried. Industry is awash with designs featuring data flow diagrams, component interface specifications, information hiding structures—the list goes on. These are helpful, unquestionably, but they lack a key quality: they do not present an integrated, all-inclusive picture of design. An architect's blueprints convey a clear picture of a house. An electrical engineer's circuit diagram shows all the parts and interconnections needed to build an electrical system. By comparison, any of the software design notations just mentioned present only a tiny part of the software's structure. (This situation probably reflects the relative age of the professions. Architects have had several thousand years to refine their craft. Electrical engineers have had over a century. Software engineers have had less than 30 years. In time, perhaps, software engineers will discover an equally descriptive notation.)

- *Resisting change.* People and industries like to stick with familiar methods and are reluctant to adopt new approaches. What project manager wants to risk her or his project's success on an unproven technology? Any technology is viewed with skepticism unless it is proven to be much more effective than current practice. New techniques necessitate training, which consumes valuable time and increases cost. A manager will readily accept change only when someone has shown that instituting the change will save time or money. Unfortunately, studies that prove such savings conclusively are few and far between in the world of software.

There are many, many more reasons why software development is so difficult. This section is by no means comprehensive; the problems discussed are only those most closely related to the *Overview of Megaprogramming Course*. Brooks (1987) gives an excellent discussion of other reasons.

## 2.3 SUCCESS STORIES IN SOFTWARE DEVELOPMENT

The preceding section might seem too gloomy. Software is involved in many aspects of our lives. In other words, people can and do develop it. So is developing software really such a problem?

The answer to this question lies more in the economics of software than in the technical problems in building it. Everyone knows how hardware costs have fallen steadily since the computer was first created. Computers keep getting faster, cheaper, physically smaller, and logically bigger. By contrast, the cost of developing software has not changed much over the past few decades. A NASA study during the late 1980s by Kouchakdijan, Green, and Basili (1989) showed the average software developer produced 24 lines of code per day, a figure about the same as in the 1950s. To be sure, today's software developers are building much larger systems. Then again, they program in much more advanced languages, use interactive terminals instead of punched cards and paper tapes, and have access to development environments which would turn a 1950s programmer green with envy. One would think that these advances should make today's developers much more productive. In general, this does not seem to be the case. Software remains costly to develop and maintain.

However, some projects have produced software at much lower cost than average. Sometimes this improvement comes from the people staffing the project: Boehm (1981) has shown that individuals' productivity varies by a factor of 4 depending on their skills. Other times, though, the difference can

be attributed to technical factors. This section explores some promising areas of the state of the art in software development.

### 2.3.1 PROGRAMMING LANGUAGES AND PRODUCTIVITY

There is some evidence that productivity is independent of the language or environment a person uses. Assume a person writing in assembly language averages 24 lines of assembly language each day. That same person would average 24 lines of Pascal each day, or 24 lines of Ada each day. If they were devising a spreadsheet, they would average 24 lines of the spreadsheet language each day. The explanation for this is that the human mind can deal with a certain amount of detail and complexity. Each of these languages has its own set of complexities the person must master.

This does not mean language is unimportant. The 24 lines of Pascal are not equivalent to 24 lines of assembly language; they are probably equivalent to several hundred assembly language instructions. Thus, the Pascal programmer will be far more productive than the assembly language programmer— that is, will build the same application quicker. The Ada programmer will be better yet, and the spreadsheet programmer will beat them all.

Of course, the spreadsheet programmer can only create spreadsheets, whereas the Pascal and Ada programmers have more options; none of them has the flexibility of the assembly language programmer. The programmers working in higher-level languages are working in restricted problem domains. That is, they are dealing less with characteristics of their computer and more with characteristics of the problem they are solving. The spreadsheet programmer's energies are directed toward creating the spreadsheet formulas. If the Pascal programmer tried to write the same spreadsheet, he or she would also have to consider details of creating and manipulating data structures to represent the spreadsheet. These details are irrelevant in the spreadsheet language; they are embedded in the spreadsheet program itself. The assembly language programmer would have to consider not only these details, but computer-specific details as well—which registers to use, the most efficient memory locations for information, and other details that are irrelevant in Pascal because they are embedded in the compiler.

Software development started with languages that forced people to consider computer-related details. It has been moving steadily away from such languages ever since. Languages like Pascal and Ada are intended to help programmers represent algorithms. They are therefore suitable first languages because computing courses begin by teaching students algorithmic programming concepts. However, one of the crucial achievements of software development is the realization that many parts of a program are best represented using nonalgorithmic constructs. Instead, programmers write software using a language specific to the domain of the problem being solved. This language is at a higher level than Pascal or Ada. Therefore, the person who writes 24 lines in this higher-level language achieves a result equivalent to the person who writes several hundred lines in Pascal.

This movement to higher-level languages has yielded significant productivity increases. The rest of Section 2.3 briefly describes examples of such languages, with insights into how they arose.

### 2.3.2 SPREADSHEETS

Spreadsheets have become increasingly popular in the last decade. They excel in expressing complex calculations. They relieve the user from having to worry about user interface arrangement, for they

provide a standard. They allow tabular data entry and present data in that and a variety of other formats, such as pie charts and bar graphs.

Spreadsheets are a simple example of programming in a restricted domain. Creating a spreadsheet is, after all, a form of programming. It requires following a process not unlike that for software development. The first task is to determine what information should be entered into the spreadsheet, what information should be displayed, and the general appearance of that information. The second task is to design formulas that calculate the results of the spreadsheet. The third task is to implement that design by creating the skeletal spreadsheet. The fourth task is to verify that the implementation works. These tasks, then, are the Requirements, Design, Code, and Test activities from Figure 2. The average spreadsheet user may not perform them with the rigor of a professional software development company, but the necessary activities are still the same.

There are, of course, many things spreadsheets cannot do. They are inherently two-dimensional and are not well suited to data in three or more dimensions. They are not intended for esoteric functions like real-time control or animation. These restrictions are deliberate. Companies began developing spreadsheets when they realized how much information in today's world lends itself to tabular presentation. Many people were writing similar programs: no matter what data they manipulated, all were working with calculations on two-dimensional data.

The inventors of spreadsheets therefore had two great insights. First, they recognized that many people were writing programs to solve similar problems: all had tables of data as inputs and, as outputs, they had that data plus calculations derived from the input, presented in forms derivable from the original two-dimensional format. Second, they realized that the calculations being made on the input data did not require the power of a general-purpose programming language, but could be made based on matrix algebra formulas and a set of predefined mathematical functions. The former insight told them what problem they needed to solve. The latter provided them with an elegant solution. They had only to write the spreadsheet program. The users of that program could write their own "programs" but did not need professional software development skills.

### 2.3.3 USER INTERFACE GENERATORS

Implementing the user interface has traditionally been one of the most difficult and time-consuming parts of software development. In one study, Boehm, Gray, and Seewaldt (1984) discovered that, on the average, over half the code in a program handles its user interface. This predated such modern advancements as mouse-driven input and graphical windows full of color icons, so the figure is probably higher now.

To complicate matters, traditional programming languages are terrible at representing details of user interfaces—and creating a user interface is all about details. Programming languages are basically one-dimensional, a long string of statements, whereas a graphical interface is two-dimensional and hierarchical. Consider the following user interface requirement, which is fairly straightforward for a person to understand:

> The interface is to be divided into two windows. Window 1 is to be 3 inches wide and 4 inches high. It will contain three buttons and one window for text entry. Window 2 is to be 1 inch wide and 5 inches high. It will contain a text label and four buttons. Window 2 is to be positioned half an inch to the right of Window 1.

. This requirement has no simple representation in a programming language. Even though unrealistically simple, it would be implemented by hundreds, if not thousands, of lines of code, when one considers the complexities of handling inputs from both a mouse and keyboard (implied by the requirements for buttons and text entry), for placing the windows, and for displaying outputs in them.

Also, the requirement is not nearly detailed enough. It says nothing about the appearance of the windows and their contents. What colors are the windows? Do they have borders? Are the mouse buttons round or rectangular? Simple decisions like these are tedious to describe as requirements and more tedious to implement in a programming language. The person writing the requirements would much prefer to use a picture like that in Figure 5, and the engineers implementing the interface would prefer a more convenient representation than what programming languages offer.
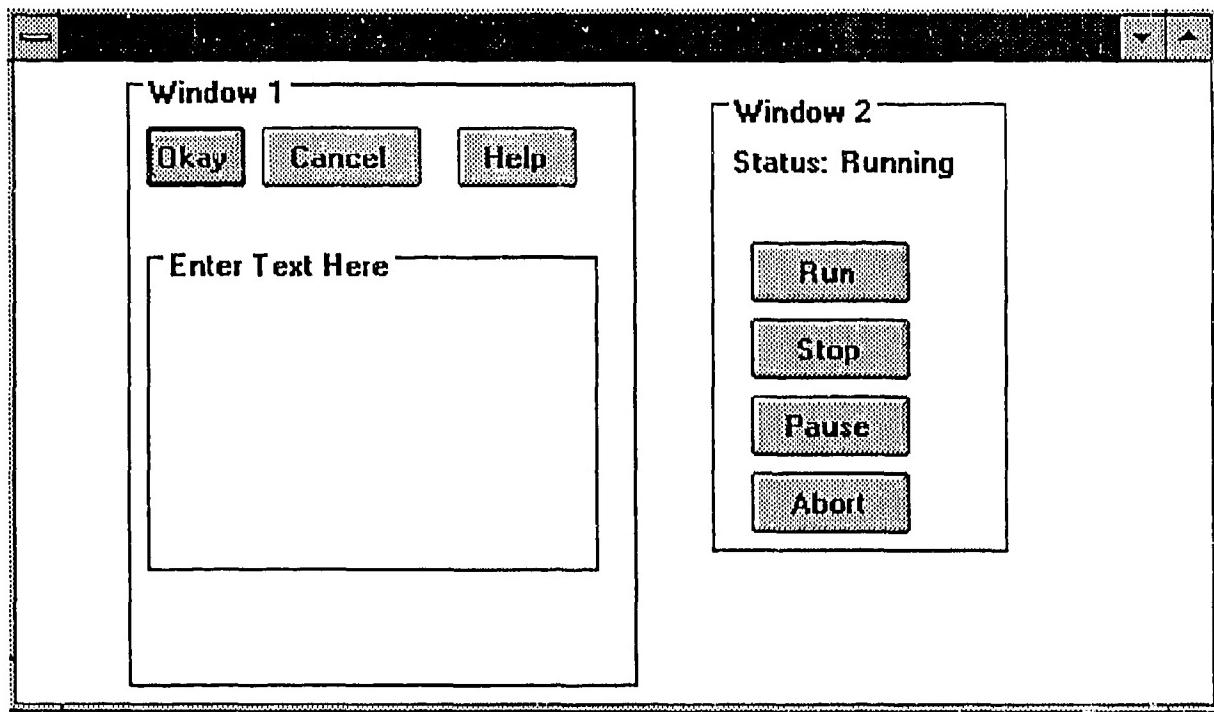


Figure 5. A Pictorial Representation of an Application's Interface Requirements

This has led people to study the domain of user interfaces. The common problem is the need to provide an effective, easy-to-use interface. People at first proposed special interface-description languages (Wartik and Penedo 1986; Hayes and Szekely 1983); but the real breakthrough came when someone realized that:

• The most natural way to describe an interface is to draw a picture of it

• A program that supports drawing a picture of an interface has enough information available to generate an implementation of that interface

The insights resulted in tools such as HyperCard, Visual Basic, TAE Plus, and many others. All are based on the principle of software developers constructing the user interface by drawing a picture of it, then writing the rest of the program in a standard programming language which uses a predefined

paradigm to obtain inputs and display outputs. (The paradigm depends on the tool and is beyond the scope of this report.)

These tools have allowed developers to build software with remarkably complex interfaces in what would once have been thought of as an inconceivably short time. The tools facilitate this because their developers studied the domain of user interfaces and discovered how to describe variations among interfaces. Their only drawback is that they inhibit flexibility. Software developers have a fixed set of input and output media (buttons, text entry areas, forms, labels, etc.). In fact, this is not really a drawback, for using the tools results in interfaces with a common "look and feel." Thus, the user of any tool created using Visual Basic (used to create Figure 5) immediately identifies the shaded rectangles as buttons and knows that pointing the cursor on one and pressing the mouse button invokes the action indicated by the button's label.

Once again, analysis of a domain has yielded an understanding of common problems and a means to state solutions to those problems in a form more natural than a programming language. For spreadsheets, the solution involved inventing a new programming language. Here, the statement is purely pictorial; no written language is necessary. Either case points toward an important trend in software development. Traditional algorithmic programming languages are intended to express algorithms; but just about any problem has nonalgorithmic aspects best expressed in a nonalgorithmic form. More and more, people are creating such forms as a means to enhance software development productivity.

### 2.3.4 RAPID PROTOTYPING

Section 2.2 described the problems that stem from improper requirements—requirements that are consistent and unambiguous, but describe the wrong problem. When a company designs and implements a solution to requirements, it makes a costly investment. If the requirements are wrong, the results can spell financial doom.

To lessen the risk of solving the wrong problem, companies sometimes rapidly create a prototype (termed a **rapid prototype**) from the requirements. They will ask a group of engineers to create, as quickly as possible, a rough version of the system. The intent is to simulate the system's external appearance (in particular, its user interface) and its functionality. The company can then use the prototype to validate whether the requirements address the intended problem. If not, the company will have avoided a large, costly mistake and will correct the requirements and the prototype until they are confident they understand the problem. They will then produce an actual version of the system. For example, URW might create a prototype litter-gathering robot with the necessary functionality but not robustness. The Park Service would test it to see whether it was viable—have they correctly described the concept of litter, or will the robot snatch a backpacker's tent?

When a company creates a prototype, they do not intend it to be of marketable quality. If they did, they would be creating an actual system rather than a prototype. The prototype is created to study specific aspects of a system, such as user interface. Generally, the prototype does not include all the functionality that the actual system will contain; a subset suffices to demonstrate the tool's utility.

Prototypes are often written in special, **very high level languages** with constructs useful for prototyping. Such languages result in small, easily modifiable programs. This is important in rapid prototyping, because prototypes must change rapidly in response to changes in requirements. (However, the programs often execute slowly; such is the price one pays for flexibility.) The languages achieve their power by incorporating constructs from the domain in question. The language for

programming Karel the Robot (Pattis 1981), used in the laboratory, illustrates this. Its purpose, from a prototyper's perspective, is to explore movement strategies. It lets the prototyper ignore such issues as:

- *How the sensors operate.* The cornfield robot needs different sensor software (and perhaps hardware) than the litter-gathering robot. Ears of corn are more or less alike and occur in expected places. Litter comes in many shapes and sizes and can be anywhere.

- *Details of motion.* The Karel programmer moves the robot using the MOVE instruction. The real robot has to account for startup inertia, maximum velocities, and many other factors.

URW might use a language such as Karel to explore issues through a rapid prototype. The prototype is based on the requirements, so the customer can study the prototype to help see whether the requirements are correct. URW and the customer use this information to revise the requirements and build a real robot.

Many interesting rapid prototyping systems exist. Some examples are IDE's Software through Pictures and i-Logix's StateMate. It's worth noting that, as computers become faster and faster, what was once an unacceptably slow rapid prototype becomes a viable software product.

# 3. AN OVERVIEW OF ARPA'S MEGAPROGRAMMING EFFORT

The United States Government is very concerned about the software development problems described in Section 2. A 1994 report to Congress (Paige 1994) stated that the Department of Defense alone had spent $30 billion on software in 1990. It estimated that the department's expenditures would jump to $42 billion in 1995. This figure does not include other branches of government, such as NASA and the Department of Energy, which have their own considerable investments. Nor does it count the commercial market in software, seen as one of the country's key assets. It is no exaggeration to say that the ability to produce quality software at a competitive price in the world market may determine a country's economic future. Hartmanis (1992) reports that software is already more than 5% of the United States' gross national product and growing—$51 billion in the corporate market alone, according to Emigh (1994).

Because it routinely produces large software systems, the Department of Defense has traditionally supported much of the country's computer science and software engineering research. Much of this research has been sponsored by ARPA. One of ARPA's most famous projects was the ARPAnet, the first major national computer network and the source for many of the ideas in today's Internet.

In 1990, to help fight the rising cost of developing and maintaining software, ARPA launched research and development of **megaprogramming**. Megaprogramming is an approach to software development that entails "building and evolving computer software component by component" (Boehm and Scherlis 1992), rather than line by line. Software developers avoid programming in the traditional way of composing lines of code into a program. Instead, they make use of existing **components**: procedures, functions, packages (in Ada), or classes (in C++). Components are the result of previous developers' efforts. Thus, each new project tries to capitalize on the fruits of earlier labors, rather than creating programs from scratch.

This is known as software **reuse**. Simply put, reuse of software means extracting pieces of existing software and using them to create new programs. Such new programs consist partly of code created expressly for the new program and partly of "reused" code. Many people today see reuse as the key to creating cheaper, higher-quality software. Using existing, already working code has clear advantages.

Reuse is not a new idea. McIlroy (1968) proposed the idea over two decades ago. Reuse may even seem an intuitively obvious strategy for software development. In fact, it has been around ever since the invention of the subroutine, which lets people reuse the same function in different places in their code.

However, reuse on a large scale—across programs, between developers, or even on a nationwide level—has been notoriously difficult. Many projects try it, only to find that reusing code results in lower productivity than creating it from scratch. The following are some of the reasons why:

- People have different programming styles. A developer is usually not content to insert a chunk of someone else's code into her or his own program. The clash between styles lowers the program's readability, which reflects poorly on the developer. Nobody wants their code to be unreadable.

- Realizing a need for reuse is one thing; finding code to fit that need is another. Establishing "reuse libraries" has proven difficult. People have tried creating classification schemes akin to those used in libraries; these schemes categorize software by function. However, these schemes often fail because developers do not know the classification schemes well enough to search for software.

  Moreover, searching a reuse library generally yields code that performs a function similar to, but not exactly matching, the need. Here the analogy to a library of books breaks down. A software engineer hopes to find and reuse a complete procedure; a scholar looks to draw material from portions of a book. A software engineer wants to use the procedure unchanged; a scholar generally recasts the material in her or his own words. As an example, a Pascal programmer who needs to sort an array of records cannot use a procedure that sorts an array of integers. The programmer can modify the procedure, but that increases the risk of introducing errors, subverting one of reuse's advantages. The problem of exactly matching needs grows with the complexity of the need and strategies for dealing with it (e.g., Ada generic parameters, C++ inheritance hierarchies) become increasingly less effective.

- Software developers have an unfortunate tendency to trust their own code and mistrust everyone else's. This attitude, termed the not-invented-here or NIH syndrome, stems from their experiences with other developers' buggy software and from their unshakable faith in the supremacy of their own programming style. Given the choice between writing something themselves or taking someone else's code and verifying that it works, they claim the former will wind up being simpler. They forget that their attitude toward others' code is the same as others' attitude toward their code. Weinberg (1971) reacted strongly against this attitude and coined the phrase "egoless programming" as the ideal that developers should adopt.

- The software world has adopted relatively few standards, and standards are what has allowed reuse to succeed in other fields. In the United States, electrical sockets deliver 120 volts of current alternating at 60 cycles per second. For this reason, televisions plug into the same sockets as microwave ovens. Their manufacturers do not have to anticipate arbitrary electrical power supplies, because the country has adopted a single standard. When a company designs an electrical appliance, its engineers can reuse an existing design for the power supply.

  By comparison, few software standards exist. There are some exceptions. The X Window System (Scheifler and Gettys 1986) provides a standard for client-server software. DOD-2167-A (Department of Defense 1988) is a standard for documentation to accompany the software development process. The systems in Section 2.3.3 define a standard look and feel for user interfaces. However, no one has yet found a standard that, when followed, helps people interconnect two arbitrary software components.

ARPA recognized these difficulties. Megaprogramming, therefore, tries a specific angle to help make reuse work. It presumes a **product line approach**. That is, organizations must consider themselves to be manufacturers of a line of software products, not producers of a single program. In fact, this usually involves only a change of attitude and not one of production. If a company sells commercial software

that runs on more than one type of computer—and most major ones do—it sells a product line and not a single product. Its engineers must make different design decisions based on the target computer. It must package the product according to the installation procedures appropriate to each computer. It must create separate documentation for each package.

A contractual company does not operate this way; it creates software for a customer's computer. However, contractual companies acquire a reputation in specific areas and do business mainly in those areas. A succession of contracts in an area makes a product-line view desirable. Suppose URW wins three contracts: one for a litter-collecting robot for the U.S. Park Service, a second for a search-and-rescue robot for the Alaska National Guard, and a third for a mineral-prospecting robot for the U.S. Department of the Interior. Although the robots will differ in many ways, they will also share many important similarities. URW's ability to capitalize on those similarities—especially by reusing software from one robot to the next—will determine how cheaply it can build each robot and, hence, how competitive it can be. In other words, if a contract-oriented company views its products as a product line over the course of several contracts, it can reuse software.

Megaprogramming demands one characteristic of software product lines: that all products in the line share similarities. This is not necessarily the standard use of the term "product line." A company in the tool business might call its home construction and repair tools a product line. This could encompass everything from a screwdriver to an air compressor. Megaprogramming employs the product line concept to speed up software development based on similarities. Similarities between a screwdriver and an air compressor are not immediately obvious. Therefore, such a product line would be of no value in the megaprogramming approach.

ARPA believes that megaprogramming holds great promise for improving America's ability to develop software. It has allocated considerable research and development funding for megaprogramming. ARPA first considered megaprogramming an advanced software engineering technique, best learned by seasoned software developers. After further consideration, however, its proponents realized that many aspects of megaprogramming were elementary and required no experience. ARPA also came to believe that merging these concepts into the early computing courses could present students with a more realistic picture of software development. Students would, therefore, be better prepared to enter the work force as skilled software developers. Even those destined for graduate studies or academic careers would benefit from knowledge of megaprogramming, for they would have a fuller understanding of the problems in software development than today's graduates.

ARPA therefore initiated the Megaprogramming Curriculum Project. Its ultimate goal is to change the computing curriculum to include megaprogramming concepts. If it is successful, students will graduate knowing how to perform megaprogramming and will think of software reuse across a product line as a natural and obvious way to develop software—quite in contrast to today's start-from-scratch mentality (see Section 5). This report and the *Overview of Megaprogramming Course* are the early products of the project: an introduction to megaprogramming concepts and a rationale for their use.

*This page intentionally left blank.*

# 4. MORE ON MEGAPROGRAMMING

So far the emphasis in this paper has been on the problems facing today's software developers. Section 3 discussed how ARPA has recognized the problems and advocated megaprogramming as a solution. Section 3 stated that megaprogramming is a product-line approach. It did not actually define megaprogramming, though, and the purpose of this section is to do so.

The lecture defines megaprogramming (see Slide 1-8) but never gives a complete definition. Indeed, the notes for Slide 4-12 end by asking students their opinions on the term's meaning. The definition in this section is deeper, not being confined to the format of slides and an accompanying page of notes. It provides a fuller explanation of the key concepts and underlying issues. The material might be too detailed for an introductory lecture, but understanding it will give greater confidence when discussing megaprogramming.

This section begins by discussing domains, a foundation of megaprogramming. It then uses domains to present the definition of megaprogramming; this is accomplished by first presenting a scenario of a company performing megaprogramming, then tying together concepts from the scenario to give an actual definition of megaprogramming. The section concludes by showing how megaprogramming improves an organization's ability to develop software.

## 4.1 DOMAINS

Section 3 mentioned that megaprogramming is a product-line approach to software development, but also pointed out that the definition for "product line" was not necessarily the standard one. In megaprogramming, all products in the line must share similarities. Understanding this concept is vital to understanding megaprogramming.

The product-line approach in megaprogramming is based on the concept of domains. Section 2 introduced domains, but informally. The definition on Slide 2-4 omits some subtleties. Here is the complete story. It is not a simple story; it requires understanding several other concepts, which will be introduced presently.

### 4.1.1 CONCEPTS OF DOMAINS

First, it may help to understand what a domain is **not**, since the word is in common use. A decidedly informal and unscientific survey of the Consortium's employees revealed that most thought domain meant "home." This meaning is not relevant to megaprogramming.

Webster's gives six definitions for the word "domain." Two are of interest:

1.  A sphere of influence or activity.

2.  The set of elements to which a mathematical or logical variable is limited, specifically the set on which a function is defined. (This is the definition used in mathematics.)

Actually, the concept of a domain in software development has little to do with the use of the word as it relates to functions. When mathematicians speak of function's domain and range, they are referring to sets of entities, such as integers, real numbers, or strings. They are concerned with the relationship between two sets. When software developers speak of a domain, they are interested primarily in the set for its own sake, not its relationship to another set. In software development, a domain has no associated range. There is no mapping from domains to anything else. Software developers concern themselves with what makes up a domain and why.

So why give the second definition? The reason is that when discussing software, people speak of domains as if they were a set of elements. Slide 2-4 mentions "the domain of robots." In the Unit 3 laboratory, students create software in the domain of robot control programs. Humans, it seems, feel a need to assign short labels to large areas. Referring to the totality of concerns would be more descriptive, but nobody has found a simple way to do it; instead, they assign a label that partially describes the domain. So software developers face the same puzzle Alice faced in Lewis Carroll's *Through the Looking Glass*:

"You are sad," the Knight said in an anxious tone; "let me sing you a song to comfort you."

"Is it very long?" Alice asked, for she had heard a good deal of poetry that day.

"It's long," said the Knight, "but very, VERY beautiful. Everybody that hears me sing it—either it brings the TEARS into their eyes, or else—"

"Or else what?" said Alice, for the Knight had made a sudden pause.

"Or else it doesn't, you know. The name of the song is called 'HADDOCKS' EYES.' "

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed. "That's what the name is CALLED. The name really IS 'THE AGED AGED MAN.' "

"Then I ought to have said 'That's what the SONG is called?' " Alice corrected herself.

"No, you oughtn't: that's quite another thing! The SONG is called 'WAYS AND MEANS;' but that's only what it's CALLED, you know!"

"Well, what IS the song, then?" said Alice, who was by this time completely bewildered.

"I was coming to that," the Knight said. "The song really IS 'A-SITTING ON A GATE;' and the tune's my own invention."

There may be many ways to label a song, but they don't necessarily describe what it IS. Instead, they describe some facet of it (all the titles are phrases from the song). Similarly, there are many ways to label a domain. The one used most often is the one describing the domain's most tangible, visible

product. For example, URW produces and sells robots, so its engineers would speak of "the domain of robots." Yet the user manuals, requirements specification, and design documents that URW produces give a fuller explanation of the domain than anyone could glean from examining URW's robots (how many people can operate a personal computer without ever consulting its user manual?). So user manuals, requirements specifications, and design documents are equally part of the domain, for they describe what is in the domain at least as well as does the set of robots URW builds.

Defining a domain as a **sphere of influence or activity** is more accurate when discussing megaprogramming (Slide 2-4 uses the phrase "well-defined area"). Unit 4 hints at the reasons why. When domain engineers define a domain, they study their company's activities to determine what programs are in the domain (Webster's second definition again). But in software engineering, a domain is more than just an arbitrary collection of programs. If two programs are to be part of the same domain, then by definition they must have some similarities. (This is another reason why definition 2 is not adequate. The domain of a function can be an arbitrary set.) Domain engineers, therefore, need criteria to determine whether two programs are related.

They derive these criteria by thinking of their company's products as solutions to the problems their customers face. To define the criteria, then. a business must first understand its customers' problems. The details of problems depend on a company's business. Some are technic  ^ hemical engineering company might develop software to control its chemical production—open. ＿ and closing valves, monitoring fluid flow, and checking pipe pressures. The software will be a morass of thermodynamic equation calculations. An aerospace company might develop satellite control software, full of navigational and positional computations.

### 4.1.2 Influence of Domain on Software Development

The technical problems mainly influence software design. Other problems are not technical and tend to exert more influence on what the software does (the difference between requirements and design, discussed in Section 2.1). Before deciding how to manufacture chemicals, a company must first decide what chemicals to manufacture. Before writing satellite control software, a company must determine what types of satellites it's going to control. In other words, a company should decide its market before it starts developing products. This decision bounds the set of problems domain engineers must define and solve.

But companies do not always use domain engineering (i.e., a megaprogramming approach) to solve problems. They only do so when they expect to set up a product line. Figure 6 helps show why. When a company first realizes the potential for sales in a **market**, it establishes a **business area**—that is, an organization tasked to conduct business in that market. This organization will study the market and determine the problems to be solved for business to succeed in that market. The organization will then propose solutions. It may decide that the most effective way to compete in the market is to offer a product line. If so, it will define the scope of the product line: the exact set of products to bring to market.

Market ---> Business Area ---> Product Line ⟨ Domain → Product Family
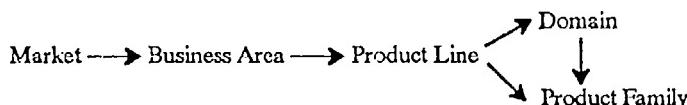
Figure 6. Relationship of Market to Domain

It must then determine the most effective way to manufacture all the products in the line. If the products all seem similar, the company will conceive of them as a domain and use domain engineering

techniques to implement **process support** (see Slide 4-2). Process support is whatever helps the company manufacture products in the product line. It consists of three parts:

- **A product family.** This is the set of programs (and related entities, like the requirements and design) that are in the domain.

- The **application engineering process** for producing products. This is what Slides 3-5 through 3-7 describe for the robot domain. The application engineering process tells how to use the product family to build products.

- **Software** to support using the application engineering process. The software for the Unit 3 laboratory is an example of such software. It references files in the directory hierarchy rooted at GEN\AC (relative to the directory where the laboratory is loaded); these files constitute the product family of robot control software.

Based on these concepts, **a domain can be defined as process support for a product line.**

Note that the lecture notes do not use the term "product family," instead referring to a domain as if it were a product family. This is a simplification. A product family is one facet of a domain. Understanding this is important to understanding megaprogramming. However, it's mainly relevant during domain engineering; it's therefore not important for the course, which covers domain engineering lightly.

As it builds the product family, the organization takes advantage of the similarities among solutions to problems in the domain. This helps it build all product-line members more efficiently than if it tried to build them all separately. This strategy only works when there are many similarities among members in the product line. The home construction tools example in Section 3 illustrates a product line that is not a product family. It cannot be studied as a domain, so building process support is not an appropriate strategy for manufacturing home construction tools.

Note that the organization does not actually build products when it builds the product family. It builds the capability to build any product in the product line. To understand this, consider an automobile manufacturing company—indeed, even a single model within a company. Even a single model has many variations, such as its color and the factory installed options. The company does not want a separate production procedure for each possible variation, though. Imagine how much more automobiles would cost if red and blue cars had to be made completely separately. Therefore, the company builds an assembly line capable of incorporating such minor variations. This assembly line is the product family. The company uses the product family (assembly line) to build products (automobiles). In this way it performs its business more effectively, making itself adaptable to a wider market.

By doing domain engineering, a company commits itself to activities in a well-defined area. This is the relevance of Webster's second definition of domain. That definition is still not wholly accurate, since it does not imply similarities, but it is conceptually significant.

In summary, defining a domain through domain engineering shows an organization how to satisfy a variety of customers in a market. That market also helps the organization understand why and how the domain will evolve in the future. And domains do evolve, just as individual programs evolve. Slide 1-4 shows Operation and Maintenance as part of the big picture of software development. It omits the simple fact that, by the time software is finally retired, the money spent on maintenance has dwarfed that spent during the initial development. Most software development occurs during maintenance.

Evolution of products has many causes. The most obvious is that companies want initial development to be as quick as possible, whereas they hope their product will be salable for a long period—all of which is termed "maintenance." Whatever the cause, it follows that software developers must be able to modify software: to redefine the problems and solutions in the domain. People redefine problems by understanding the market, predicting changes in it, and defining how products should evolve in response to those changes.

## 4.2 DEFINITION OF MEGAPROGRAMMING

Now that domains have been explained, it is time to present a definition of megaprogramming. This section will give a better feeling for what megaprogramming is and how it affects an organization.

### 4.2.1 A MEGAPROGRAMMING SCENARIO

Consider URW again. Section 2.1 presented contractual and commercial software development scenarios for URW. The practices in these scenarios are typical of how many companies develop software today. Section 2.2 discussed reasons why URW's practices might cause difficulties. Suppose URW adopts megaprogramming. What will it do differently?

Many of URW's problems stem from the independence among its projects. The two projects in Section 2.1 operated as if they were parts of different companies. They did not share information—at least, not in any formal, documented manner that could be presented in the scenarios. This is despite the many similarities that probably exist between the two projects. The robots may be different, but they are part of the same domain. So URW decides that its product line is a product family and opts to build process support to help it manufacture robots more efficiently.

To achieve this, URW must reorganize itself into something resembling Slide 4-12. URW will create a separate domain engineering group. This group is responsible for creating, monitoring, and improving the process support. Each time URW receives an order for a robot, it will start an application engineering project, as before. The difference is that the application engineering project will use the process support. Using it entails following a special, domain-specific application engineering process like that in Slide 3-6, not a general one like in Figure 2.

The course laboratory illustrates application engineering when following such a process, so it won't be discussed here. The important point is that it's a process tailored to the domain. Application engineers develop software mainly by thinking about problems in the domain. They think about such problems in any software development process—in Section 2.1, they developed a software requirements specification, which amounts to the same thing. However, the application engineers using process support don't develop the complete requirements specification. They only describe how the family member they want to build differs from other family members. This is a small part of the whole specification, so URW saves time and effort. Furthermore, the application engineers' difficult work ends once they have described the problem. The rest of software development is mechanical, derived entirely from the problem statement.

This strategy works, as the notes for Slide 3-3 mention, because application engineers are experts in the domain. They possess an intuitive understanding of the properties common to all family members. They understand, or are able to determine, the implications of a variation among family members. For this reason, Unit 3 tries hard to make students semiexperts in the robot domain prior to their laboratory experience.

It's worth noting that, in reality, URW's application engineers could not generate complete, working software automatically based on their problem statement. The process support is only as good as URW's domain engineers' predictive abilities. Customers will almost certainly demand variations URW had not anticipated. Toshiba, a Japanese company that builds power plants, instituted a form of megaprogramming and found they could generate about 70% of their software automatically. Since each power plant required over one million lines of code, they reaped incredible savings—but they still had to develop 300,000 lines of code for each new project. URW's engineers would face similar situations. To keep the introduction to megaprogramming simple, the course's laboratory glosses over this point and generates 100% of the software for the students.

Application engineers follow a domain-specific process. Domain engineers do not. They use a software development process quite different from either the application engineering process or the one in Figure 2. The objective of this process is to create, field, and enhance a product line—what the slides term process support. Figure 7 shows this process; the text following Figure 7 explains it, activity by activity.

- *Domain Management.* URW begins domain engineering by starting with the Domain Management activity. Here, URW decides what process support it wants to build. This is a critical business decision. It determines what markets URW will try to capture. The result of this decision, expressed as the domain plan, determines URW's business direction for the next several years. The domain plan guides domain engineers in all their other activities (hence the nested boxes).

- *Domain Analysis.* URW begins the Domain Analysis activity after making the decision on what process support to build. The goal of Domain Analysis is to produce a specification of the production line. URW will task a group of experts in the domain of robots (domain engineers) to study the problems that robots must solve and to uncover the similarities and differences among these problems. The domain engineers will use this information toward two ends:

  — To describe the application engineering process.

  — To specify the properties of a solution to any of the problems in the domain. That is, they have described a family of problems. They describe a family of solutions, one solution per problem, and describe the relationship between the two families. Thus, if an application engineer describes a problem, the domain engineer has provided a means to identify a solution.

- *Domain Definition.* Domain engineers produce the specification in two steps. During the first, the Domain Definition activity, they produce an informal description of the domain.

- *Domain Specification.* Domain engineers use the domain definition during the Domain Specification activity, when they produce the more rigorous domain specification. The domain definition is deliberately short on details; the domain specification is sufficiently precise to serve as a requirements specification. The domain definition is small and focuses on concepts; the domain specification is much larger and lets a domain engineer answer any question about the problems and solutions in the domain. This paradigm of refining an informal description is a common engineering design strategy.

This has briefly described the process domain engineers follow to create the products shown in Slides 4-3 through 4-10. Note that the domain engineers have not created any software at this point.

They have only described problems that software they create must solve, and have described the architecture of proposed solutions.
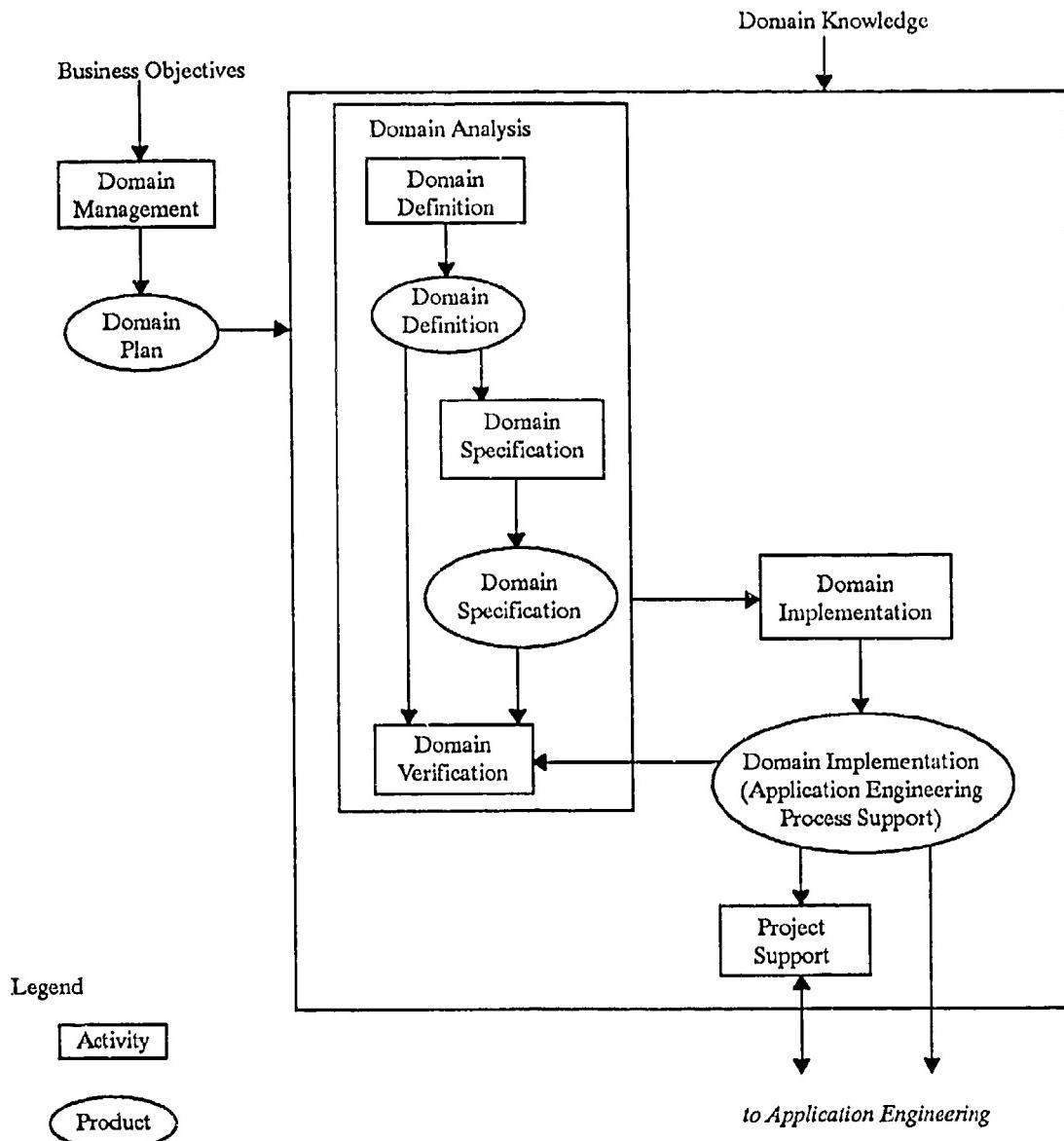


Figure 7. The Domain Engineering Process

- **Domain Implementation.** With the domain specification in hand, domain engineers begin the Domain Implementation activity. Here, they build the process support that application engineers will use. This involves building the product family, defining the application engineering process, and developing the software to assist in process support. In more detail:

    - The product family consists of reusable software components. These are based on the architecture developed during domain analysis. The domain engineers will implement a software component for each box in the architecture (see Slide 4-6). Note that the architecture itself is not part of the software, any more than a building's architecture is part of a building.

- The application engineering process describes the process for specifying requirements (precisely stating the problem, see Slide 4-5) and the generation procedures (see Slide 4-11). Generation procedures are an exact description of the relationship between a problem and its solution. Application engineers, once they have stated a problem, will use the generation procedures to produce the software.

- Developing the software that automates the process support is a miniature software development process in its own right. URW tries to determine which activities of the application engineering process are most likely to introduce errors. It then studies those activities and, where practical, automates them. In the laboratory, adapting the reusable software components to satisfy a particular problem statement is one of the most difficult tasks for humans to perform. It is not a complex task, just a tedious one—describing how to do it manually would sometimes require hundreds of instructions. It has therefore been automated.

- *Domain Verification.* Domain engineers then determine whether the process support satisfies the domain specification. This is analogous to testing in a conventional project, but domain engineers must see if each problem in the domain has a correct solution, rather than checking a single solution against a single problem.

- *Project Support.* In addition to these activities, the domain engineering group acts as a support organization with respect to application engineering projects. Domain engineers are responsible for setting up the production line and helping application engineers use it. They are also responsible for noting, and correcting, deficiencies in the process support.

The result of all this is that URW has launched a set of concurrent **iterative processes**. Domain engineering is not a simple progression of steps, as in Figure 2. The results of domain engineering feed back into the initial activities of domain engineering as domain engineers evolve the domain. Thus, each domain engineering **iteration** results in improved process support. URW can better position itself for future sales as a result of this effort. At the same time, an application engineering project responds to its changing customer needs—whether that of an individual customer or of an entire marketplace—by refining its problem statement and building new, improved versions of a product. Each application engineering iteration results in a new product.

### 4.2.2 What Is Megaprogramming?

URW has used megaprogramming to develop products. That is, it has achieved a capability to sell a variety of products in a changing marketplace by doing the following:

- Appealing to a wide customer market by thinking in terms of a product line rather than a single product

- Realizing that there are many similarities among its products and taking the time to study these similarities

- Exploiting these similarities by implementing process support

When a company takes these three actions, it is practicing a megaprogramming approach to software development.

There are four key concepts in megaprogramming:

- *Product Families.* A family is a set of things that are sufficiently similar that it is worthwhile to understand the common properties of the things before considering special properties of specific instances. Domain engineers think in terms of product families, rather than in terms of individual products. By doing so, they anticipate a range of needs. This helps a company like URW plan for the future as well as the present. It has helped companies like Microsoft Corporation make such products as Word and Excel work on a range of computers. (That is, there are several versions of Word, each of which operates on a different platform. Each version is a product. The set of all versions is a product family.) It's also important to someone developing software for their personal computer as they strive to understand the nature of the problem at hand.

- *Iterative Processes.* Any organization, software or otherwise, should expect to use iterative processes. Experience is the best teacher, says the old saw, and has proven the only reliable way to diagnose trouble spots in a production capability. (Henry Ford first had lines of workers moving between stationary cars, rather than cars moving between stationary workers.) Quality software is inevitably produced only through successive iterations of specification, design, and implementation activities. The company or individual that uses an iterative process recognizes the difficulty of writing the requirements for a useful product without having built and used it, or of keeping up with ever-changing technology. Companies that write operating systems understand this need very well. As of this writing, Microsoft Corporation has produced six versions of DOS, Apple Computer seven versions of its Macintosh operating system.

- *Specification.* A specification is a precise description of the properties needed of some entity, such as a program. Of particular interest are specifications of requirements. If the requirements specification really is precise—not an informal description in English, but something akin to the set of decisions in the laboratory—then it can be used as the input to a generation procedure, from which software can be generated automatically.

- *Reuse.* Constructing software from existing components reduces effort and increases reliability, two of the great software engineering problems. This seemingly simple panacea is made complex by the difficulty in integrating components drawn from arbitrary sources. Domain engineering, and the standardization that results from it, greatly increases the ability to reuse existing software components.

Megaprogramming stresses engineering concepts. Much of its power comes through instituting standards—as Section 3 mentioned, other disciplines are far ahead of software engineering in doing so. The analysis of similarities during domain engineering is, in effect, specifying standards for the domain: standard requirements (what is common to all problems in the domain), standard designs (the architecture in Unit 4 shows what is standard to all designs in the domain), and standard implementations (components shared among all robots establish standard algorithms).

### 4.2.3 PERSPECTIVES ON MEGAPROGRAMMING

The original ARPA definition of megaprogramming stresses the product-line approach to software development. The description in this section is but one way to achieve a product line. It is based on a software engineering approached called Synthesis, developed by the Consortium. Synthesis has been

used successfully on several industrial projects. More information is available in the *Reuse-Driven Software Processes Guidebook* (Software Productivity Consortium 1993).

Other researchers have taken a different approach to megaprogramming. Space limitations preclude discussing them here (see STARS [1992]).

## 4.3 BENEFITS OF PRACTICING MEGAPROGRAMMING

Section 2.2 described five significant software development problems. Megaprogramming will not always solve the problems, but practicing it can significantly alleviate many or all of them. This section explains how megaprogramming helps organizations overcome each problem.

- *Expressing software requirements.* Application engineers practicing megaprogramming have two advantages. First, because they describe a problem in terms of how it differs from other problems in its domain (see the laboratory), they reuse requirements instead of having to write them from scratch. They have fewer choices to make and, thus, are less likely to make an incorrect choice. Second, the requirements they create are precise enough to allow rapid prototyping (this is essentially what goes on in the course laboratory: the students create and execute a rapid prototype of the robot's control software). This means the application engineers can show a model of the system to customers at the time the requirements are written, rather than having to wait until after the system is implemented. Customers can immediately point out deficiencies and misunderstandings. Consider Slide 1-6: fixing a requirements error during testing costs four times as much as fixing it during requirements. Practicing megaprogramming helps application engineers find errors during requirements.

- *Following a software development process.* Much of this difficulty stems from the generality of the waterfall software development process so widely used today. This process gives only general guidance and does not direct the day-to-day activities. The reason it is so general is that adding more detail requires making assumptions about the type of software being developed—in other words, about the domain. Domain engineers are tasked to add exactly this information to the process they create for application engineers. Application engineers, therefore, have a very detailed process to follow.

- *Keeping requirements and documentation up to date.* This problem often arises not so much from bad intentions as from neglect: no one is explicitly tasked to do the job. An organization that practices megaprogramming explicitly recognizes the need to keep requirements and documentation consistent with code. Domain engineers are responsible for developing and maintaining standards for the domain. This includes documenting the standards and keeping them up to date.

- *Grasping software design.* Megaprogramming helps here by placing the design in a domain-specific context. Often, understanding a design is simply a matter of evolving a standard, common terminology and set of descriptive techniques. These tend to be a natural by-product of multiple iterations of domain analysis.

- *Resisting change.* Much of the strategy in domain engineering involves planning for change, attempting to anticipate and soften its effects. Of course, industry must be prepared to adopt megaprogramming, which is in itself a major change. However, organizations that have tried and stuck with it report improved productivity (O'Connor et al. 1994).

# 5. THE NEED FOR MEGAPROGRAMMING IN HIGH SCHOOLS AND UNIVERSITIES

The previous sections have shown megaprogramming's potential importance as an industrial software development approach. Industrial practices are not necessarily suited to classroom settings, however. Many megaprogramming details are relevant to a large corporation but not in a classroom. These details arise because corporations engage in huge multiyear projects, which are simply not practical for students.

Nevertheless, the Megaprogramming Curriculum Project believes teachers should introduce megaprogramming into their courses. This section will explain why.

## 5.1 THE CURRENT CURRICULUM: STRENGTHS AND WEAKNESSES

Section 1.1 discussed the current computer science curriculum briefly. Understanding megaprogramming's importance in academia requires a more thorough analysis.

The discussion that f    vs concentrates on the first computing course. This course's content sets the stage for students' renaming education in computing and therefore plays a fundamental rôle. Subsequent courses—data structures, operating systems, etc.—build on its content. Changing it necessitates changing the rest of the curriculum. It therefore deserves special study.

A survey of model curricula (Larson and Stehlik 1990; Tucker et al. 1991; Merritt et al. 1993) shows that the first course emphasizes the following topics:

- *Programming Language Notation.* To write a program, a student must master the nuances of a programming language's syntax and must feel comfortable expressing algorithms using that syntax.

- *Algorithms.* A student learns sequencing, conditional execution, and iteration very early in the course—sometimes within the first two weeks, if teachers are using a special language such as Karel the Robot. A student spends much of her or his first year learning specific algorithms to be used in conjunction with recently introduced language features. For example, many teachers introduce sorting algorithms almost as soon as they introduce arrays.

- *Data Structures.* A student learns about arrays, records, pointers (less frequently), and how to use them to create certain simple abstract structures, such as trees or stacks.

- *Programming Methodology.* A student learns approaches to software specification, design, and verification. The coverage of specification is necessarily brief because students lack the mathematical background necessary to grasp most specification techniques. The emphasis

during design is on top-down decomposition methods, such as structured programming and information hiding. During verification, students learn testing techniques and an informal version of axiomatic verification.

This emphasis on programming methodology does not extend to teaching software process concepts, despite the obvious overlap. Several of the model curricula present software design concepts, but none expect students to separate design from implementation.

These topics, as Section 1.1 mentioned, have formed the basis of the introductory computing course almost from the very beginning. To be sure, the course's content has not been static. Context-free grammars let teachers explain syntax quickly and precisely. Algorithms were once presented as flowcharts (the effect of which was mainly to make students grasp a second notation); now teachers use structured programming concepts. Programming methodology is now understood much better: teachers can show how stepwise refinement leads from requirements to a workable design.

The Megaprogramming Curriculum Project believes these topics are still valid—writing software requires mastering a formal notation and using it to express algorithms. However, the project questions the emphasis placed on them, as opposed to certain other topics, and the manner in which they are introduced.

Paradoxically, the reasons why stem in part from the course's strength: students who take it are able to write simple programs in an arbitrary application area. This is an excellent model for a high school or university. Instructors in areas like chemistry, physics, or civil engineering—not to mention computer science—can assign their students computationally intensive problems after the students have had only a single course in computing. Indeed, the student who knows sequencing, conditional execution, and iteration can construct an algorithm to solve any solvable problem—a well-known theoretical result in computer science (Böhm and Jacopini 1966). Few other disciplines offer such a general and powerful introductory course!

This generality comes at a price. Students completing the first course have the following perspective:

- *They place too much emphasis on the programming language they have learned.* This is a natural consequence of their struggle to learn a new notation. A language like Pascal has a complex, unforgiving syntax. One of their first major problems is learning that syntax, so they attach importance to the language syntax rather than the semantics—that is, the ease of expressing algorithms in it. By contrast, the skilled software developer can learn the syntax of any programming language in a day or two.

- *They think of a program as an algorithm.* Experienced software developers, however, realize that a program consists of a set of modules. They use modules as the basis for contemplating and expressing software design; designs of programs based on algorithms have not proven satisfactory. Because engineering is not possible without a design, software development without a design will never be a disciplined activity.

- *They think of software development as consisting of coding and testing.* This paper has discussed the steps of real software development in depth and has shown them to be far more complex than just these two steps. Experienced software developers see coding as a largely rote activity that follows design. Furthermore, perceiving the cyclic nature of software development, developers recognize maintenance and the problems it causes and try to plan for it.

- *They lack a scientific basis to analyze the software they produce.* Most high schools and universities teach a course called "Introduction to Computer Science," but they teach little or no science during the course. Roughly speaking, computer science enables someone to assess the quality of a computer program. Here, quality can be defined in terms of many factors: execution speed, memory use, user interface friendliness, reusability, and degree of fidelity to the requirements, to name a few measures. Most introductory courses only teach students to test their software, and not in any systematic way. Other issues are ignored. Most students never practice a disciplined, scientific approach to software development. Experienced software developers try to be as disciplined as they can, scientifically analyzing their software design (**not** implementation).

This discussion should not be taken to mean that the current content of introductory courses is easy. Formal syntax concepts and algorithmic problem-solving are difficult skills and require time to master. The difficulty is making sure that students do not see these as the only obstacles to writing software, or even the main obstacles. People with this perspective grasp only a small part of what software development is all about and require extensive retraining to be successful in the work force.

Even given these problems, it's worth questioning whether to make any changes to a curriculum that does a generally good job of serving the community. The Megaprogramming Curriculum Project believes that the current model has its place, but mainly for a service course. Those students who intend to become professional software developers (or even computer science researchers) would benefit from a curriculum that teaches more about how software is actually developed.

## 5.2 BENEFITS OF MEGAPROGRAMMING FOR STUDENTS

The Megaprogramming Curriculum Project believes students will benefit from learning megaprogramming in four ways:

- They will learn to see software development as an exercise in science and engineering. This paper has shown how megaprogramming fosters a disciplined approach to software development. Students who learn megaprogramming will be able to produce better software—and will be able to back up claims of quality.

- They will be able to work on more complex systems. Megaprogramming encourages reuse, so students can reuse existing software that performs complex functions. The laboratory in the *Overview of Megaprogramming Course* demonstrates this point. Students create reasonably complex robot control software by reusing existing software components. Examples of this nature are more exciting than typical student projects; as a result, students' motivation will be increased.

- They will learn more about what software development is really like. This makes them better prepared for their careers.

- Their perspective toward software development will be holistic. Megaprogramming encompasses all aspects of software development. The instructor can introduce any software-related topics, and can tie them together. Most schools teach at least some software engineering topics. (e.g., programming methodology, as discussed in Section 5.1) but, lacking a complete approach such as megaprogramming, do not let students understand the full importance of the topics.

## 5.3 WHY IN THE FIRST COURSE?

M ny undergraduate institutions teach megaprogramming concepts. Students often take a software engineering course in their third or fourth year and learn about software processes, software architecture, etc. The Megaprogramming Curriculum Project believes that these concepts should be introduced in the first course, even in high schools. Is this really the correct place for them?

A survey of recent conferences in software engineering education (e.g., Díaz-Herrera 1994) shows a trend toward teaching software engineering concepts earlier. Educators complain that, in the current curriculum, students' early education appears to stress the wrong skills. Their abstraction abilities—so crucial to expressing and communicating specification and design concepts—are poorly developed. Their software development techniques are not suited to real projects. As a result, one of the software engineering course's major goals is to make students forget the improper skills they have acquired.

Some secondary considerations support introducing megaprogramming early. Interest in computer science as a major dropped considerably during the past decade (Gries and Marsh 1989). One solution to this is to provide students with more interesting problems. As discussed in Section 5.2, megaprogramming increases the size and complexity of the software students can be expected to create.

The Megaprogramming Curriculum Project may also been seen as an experiment in finding the correct amount of theory to introduce in the first course. Computer science educators have always been interested in this. The University of Maryland is a noteworthy example (Mills et al. 1989). Their first course introduces students to the more mathematically oriented aspects of computer science and software engineering, such as axiomatic program definition and formal program verification.

The Megaprogramming Curriculum Project is striking a balance between this extreme and the mainstream approach. In megaprogramming, the science comes more from the application domain than from the pure mathematics associated with computer science. The student therefore learns to see rigorous, quantitative analysis as a natural part of software development. However, the student needs no special mathematical background.

## 5.4 BENEFITS OF TEACHING THE *OVERVIEW OF MEGAPROGRAMMING COURSE*

Now that the benefits to the students have been explained, it remains to discuss what the instructor may expect from teaching the *Overview of Megaprogramming Course*. This course, as the name implies, is not a complete course in megaprogramming. After taking the course, the student:

- Is an application engineer, for the domain of robot control software

- Has an appreciation of the important issues in industrial software development

- Has learned the concepts of megaprogramming, a new and exciting approach to developing software

These concepts include application engineering and domain engineering. The course teaches the student how to perform application engineering in a particular domain. It deliberately omits any discussion of how to perform domain engineering. This complex topic is beyond the scope of an overview course. (The Megaprogramming Curriculum Project hopes to create, or encourage

instructors to develop, subsequent course units that will teach students how to perform domain engineering.)

Despite the course's brevity, the instructor who teaches the course should find that her or his students have a more realistic understanding of the role software plays in complex systems and of the difficulties in developing it. By using a hypothetical company, and by providing considerable detail on that company, the course tries to show all of the major considerations that influence software. Often, these considerations are not technical but, as the laboratory shows, economic. Nor do they always have obvious, clearly defined answers. Software practitioners must be able to justify their decisions. The *Overview of Megaprogramming Course* provides exercises to help them realize this.

The Megaprogramming Curriculum Project also hopes that teaching the course will help instructors become more aware of the states of the art and practice in industry. The course, including this document, serves as a sort of liaison between industry and academia.

Teachers who have taught the course have reacted positively toward it. Some have begun to incorporate its concepts into their regular materials. The Megaprogramming Curriculum Project is pleased by this initiative, for it shows that the teachers consider the material valuable and a fundamental part of a student's education. The project hopes that, through actions such as this, megaprogramming materials will continue to influence the computing curriculum.

*This page intentionally left blank.*

# APPENDIX A. RELATION OF LECTURE SLIDES TO THIS REPORT

This report makes many references to slides from the lectures. Someone preparing to lecture on this material may appreciate inverse references: which sections of this report explain the material of a given slide. Table 1 presents that information.

Table 1. Mapping of Slides to Sections in This Report

| Slide | Section(s) in Report | Slide | Section(s) in Report |
|-------|----------------------|-------|----------------------|
| 1-2   | 2.1                  | 3-5   | 4.1, 4.2.1           |
| 1-3   | 2.1, 2.2, 2.3        | 3-6   | 2.2                  |
| 1-4   | 2.1                  | 3-7   | 4.2.1, 4.2.2         |
| 1-5   | 2.1                  | 4-2   | 4.2                  |
| 1-6   | 2.2                  | 4-3   | 4.1, 4.2.1           |
| 1-7   | 2.1                  | 4-4   | 4.1, 4.2.1           |
| 1-8   | 4.2                  | 4-5   | 4.2.1, 4.2.2         |
| 2-2   | 4.1                  | 4-6   | 4.2.1, 4.2.2         |
| 2-3   | 4.1                  | 4-7   | 4.2.1, 4.2.2         |
| 2-4   | 2.3.1, 4.1           | 4-8   | 4.2.1, 4.2.2         |
| 2-5   | 3, 4.2               | 4-9   | 4.2.1, 4.2.2         |
| 2-6   | 4.1, 4.2, 4.3        | 4-10  | 4.2.1, 4.2.2         |
| 3-2   | 2.1, 4.2.1           | 4-11  | 4.2.1, 4.2.2         |
| 3-3   | 4.1, 4.2.1           | 4-12  | 4.2.2                |
| 3-4   | 4.1, 4.2.1           |       |                      |

*This page intentionally left blank.*

# APPENDIX B. STRUCTURE OF THE *OVERVIEW OF MEGAPROGRAMMING COURSE*

The *Overview of Megaprogramming Course* is organized into four units and takes approximately 1 to 2 weeks to cover. This appendix briefly describes each unit.

## B.1 UNIT 1: SOFTWARE DEVELOPMENT

The first unit discusses how today's companies develop software. It shows data that justifies the points it makes about what software development is really like. It covers important software development concepts, mainly process and requirements.

This unit uses the concepts of process and requirements to motivate the need for reuse and megaprogramming. A simple chart of the megaprogramming process helps students to put the rest of the course in the proper context. An in-class exercise asks the students to try writing complete requirements for simple, everyday problems, thereby showing them how difficult the requirements step is. This unit introduces the vending machine example, which will be used throughout the course. For their homework assignment, the students devise a set of requirements for the vending machine.

## B.2 UNIT 2: CONCEPTS OF MEGAPROGRAMMING

Unit 1 covered general software development concepts. This unit introduces concepts specific to megaprogramming.

It presents software development as a process of analyzing a problem and implementing a solution. It then defines domains, shows how domains support reuse, and discusses domain engineering and application engineering. In-class exercises for this unit have the students identify whether or not simple, everyday classes of applications are domains. For the vending machine problem, the students combine their requirements, identify similarities and differences among their different vending machines, and identify those components that can work for all vending machines. For homework, students identify what components they need for their own vending machine.

## B.3 UNIT 3: APPLICATION ENGINEERING

This unit shows students what software development is like when using megaprogramming. The unit introduces problems associated with robot control software. Students learn how megaprogramming helps them produce such software.

At the end of this unit, the students are given a laboratory exercise. They must carry out application engineering in the robot control software domain. They are taught about URW, the hypothetical

company from Section 2. The students build the software for three customers: a farmer who needs corn harvested, a representative from the Alaska National Guard who wants search-and-rescue robots, and a National Park Service ranger who wants robots that can pick up litter. The application engineering process the students follow shows them the commonalities among the software requirements for these seemingly disparate robots—although it also calls their attention to the specific differences. The software solutions they generate are constructed purely from reusable components. They integrate these components according to a domain-specific software architecture and can, therefore, see the similarities and differences among implementations as well. Robots in this domain are all similar in that they search autonomously for some type of object. However, they differ based on such characteristics as the terrain they search (field, tundra, or forest), the types of objects for which they search (corn, people, or litter), the actions they take on finding an object (pick up and return, locate only, continue indefinitely), their search strategy (zigzag or sweep), and their initial amount of energy (in joules). The application engineering process asks the students to reason about robots in terms of these concepts. Students must also make quantitative comparisons of robots based on a cost model that is provided and an execution environment that simulates the time and energy needed to perform a mission. Students vary certain quantities and see the relative effects on a robot's cost and the time needed to complete a mission.

The laboratory exercise is built on top of the Karel-the-Robot concepts developed by Pattis (1981). A front end to an existing Karel implementation asks the students to make decisions that differentiate one robot from other robots within the domain as described in the previous paragraphs. Based on the decisions, the students then use the environment to generate robot software from the reusable components within the domain and to simulate the robot moving through the specified terrain performing the specified mission.

## B.4 UNIT 4: DOMAIN ENGINEERING

This unit discusses what a company must do to achieve the software development capability the students saw in Unit 3, defining this as domain engineering. Because of time constraints, the unit covers what but not how.

The unit covers the information domain engineers use to define a domain and aspects of the support domain engineers provide for the application engineer. The exercise for this unit helps the students see how they can use megaprogramming in the development of vending machines. An examination evaluates their mastery of the material.

# LIST OF ABBREVIATIONS AND ACRONYMS

ARPA  Advanced Research Projects Agency

NASA  National Aeronautics and Space Administration

URW  United Robot Workers, Inc.

*This page intentionally left blank.*

# REFERENCES

Backus, John
1978

The History of FORTRAN I, II, and III. *SIGPLAN Notices* 13, 8:165–180.

Boehm, Barry
1981

*Software Engineering Economics.* Englewood Cliffs, New Jersey: Prentice-Hall.

Boehm, Barry, Terence Gray, and Thomas Seewaldt
1984

Prototyping vs. Specifying: A Multi-Project Experiment. *IEEE Transactions on Software Engineering* SE-10, 3:290–302.

Boehm, Barry, and William Scherlis
1992

"Megaprogramming." In *Proceedings, DARPA Software Technology Conference.* Los Angeles, California.

Böhm, Corrado, and Guiseppe Jacopini
1966

Flow Diagrams, Turing Machines and Languages with only Two Formation Rules. *Communications of the ACM* 9:366–371.

Brooks, Frederick
1987

No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* 20, 4:10–19.

CACM
1989

A Debate on Teaching Computer Science. *Communications of the ACM* 32:1397–1414.

Coad, Peter, and Edward Yourdon
1990

*Object-Oriented Analysis.* Englewood Cliffs, New Jersey: Prentice-Hall.

Datamation
1994

NIH Syndrome Strikes Again. *Datamation* 40, 11:90.

Department of Defense
1988

*Military Standard: Defense System Software Development* (DOD-STD-2167A). Washington, D.C.: Department of Defense.

Díaz-Herrera, Jorge
1994

*Software Engineering Institute 7th Conference on Software Engineering Education.* New York, New York: Springer-Verlag.

Emigh, Jacqueline
1994

Software Forum—Corporate Market to Reach $51B in '94. *Newsbytes* April 25, 1994.

Gries, David, and Dorothy Marsh
1989

The 1987–1988 Taulbee Survey. *Communications of the ACM* 32, 10:1217–1224.

Hartmanis, Juris
1992

Computing the Future. *Communications of the ACM* 35, 11:30–40.

Hayes, Philip, and P. Szekely
1983

Graceful Interaction through the Cousin Command Interface. *International Journal of Man-Machine Studies* 19, 3:285–305.

Humphrey, Watts
1989

*Managing the Software Process.* Reading, Massachusetts: Addison-Wesley.

Ince, Darrel
1988

*Software Development: Fashioning the Baroque.* New York, New York: Oxford University Press.

Kouchakdjian, Ara, Scott Green, and Victor Basili
1989

"Evaluation of the Cleanroom Methodolog" in the Software Engineering Laboratory." In *Proceedings of the Fourteenth Annual Software Engineering Workshop.* Greenbelt, Maryland.

Larson, Barbara, and Mark Stehlik
1990

*Teacher's Guide to Advanced Placement Courses in Computer Science.* New York, New York: Educational Testing Services.

Marco, David, and Clement McGowan
1987

*SADT: Structured Analysis and Design Technique.* New York, New York: McGraw-Hill.

McCracken, Daniel, and Michael Jackson
1982

Life Cycle Concept Considered Harmful. *Software Engineering Notes* 7, 2:29–32.

McIlroy, Douglas
1968

"'Mass-Produced' Software Components." In *Proceedings, NATO Software Engineering Workshop.*

Merriam
1977

*Webster's New Collegiate Dictionary.* Springfield, Massachusetts: G.&C. Merriam Company.

Merritt, Susan, Charles Bruen, Philip East, Darlene Grantham, Charles Rice, Viera Proulx, Gerry Segal, and Carol Wolf
1993

ACM Model High School Computer Science Curriculum. *Communications of the ACM* 36, 5:87–90.

Mills, Harlan, Victor Basili,
John Gannon, and
Richard Hamlet
1989

Mathematical Principles for a First Course in Software Engineering. *IEEE Transactions on Software Engineering* SE-15, 3:550–559.

O'Connor, James,
Grady Campbell,
Catharine Mansour, and
Jerri Turner-Harris
1994

Reuse in Command-and-Control Systems. *IEEE Software* 11, 5:70–79.

Paige, Emmett Jr.
1994

*DoD Software Reuse Initiative.* Report to Congress, March 1994. Washington, D.C.: Department of Defense.

Pattis, Richard
1981

*Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal.* New York, New York: John Wiley and Sons.

Prey, Jane, James Cohoon, and
Greg Fife
1994

"Software Engineering Beginning in the First Computer Science Course." In *Proceedings, Software Engineering Institute 7th Conference on Software Engineering Education.* New York, New York: Springer-Verlag (359–374).

Royce, W.
1970

"Managing the Development of Large Software Systems: Concepts and Techniques." In *Proceedings, WESCON.*

Scheifler, Robert and
Jim Gettys
1986

The X Window System. *ACM Transactions on Graphics* 5, 2:79–109.

Software Productivity
Consortium
1993

*Reuse-Driven Software Processes Guidebook.* SPC-92019-CMC, version 02.00.03. Herndon, Virginia: Software Productivity Consortium.

STARS
1992

*On the Road to Megaprogramming.* Arlington, Virginia: STARS Technology Center.

Tucker, Allen, Bruce Barnes,
Robert Aiken, Keith Barker,
Kim Bruce, Thomas Cain,
Susan Condry, Gerald Engle,
Richard Epstein, Doris Lidtke,
Michael Mulder, Jean Rogers,
Eugene Spafford, and
Joe Turner
1991

*Computing Curricula 1991: Report of the ACM-IEEE-CS Joint Curriculum Task Force.* New York, New York: ACM Press.

Wartik, Steven, and
Maria Penedo
1986

FILLIN: A Reusable Tool for Form-Oriented Software. *IEEE Software* 3, 2:61–69.

Weinberg, Gerald
1971

*The Psychology of Computer Programming.* New York, New York: Van Nostrand Reinhold.